# IMPLEMENTASI *FAILOVER* DAN *AUTOSCALING* KONTAINER *WEB SERVER* NGINX PADA DOCKER MENGGUNAKAN KUBERNETES

**SKRIPSI**

Untuk memenuhi sebagian persyaratan
Memperoleh gelar Sarjana Komputer

Disusun oleh:
Yosua Tito Sumbogo
NIM: 145150200111050

PROGRAM STUDI TEKNIK INFORMATIKA
JURUSAN TEKNIK INFORMATIKA
FAKULTAS ILMU KOMPUTER
UNIVERSITAS BRAWIJAYA
MALANG
2018

# PENGESAHAN

IMPLEMENTASI *FAILOVER* DAN *AUTOSCALING* KONTAINER *WEB SERVER* NGINX PADA DOCKER MENGGUNAKAN KUBERNETES

SKRIPSI

Diajukan untuk memenuhi sebagian persyaratan
memperoleh gelar Sarjana Komputer

Disusun Oleh :
Yosua Tito Sumbogo
NIM: 145150200111050

Skripsi ini telah diuji dan dinyatakan lulus pada
3 Agustus 2018
Telah diperiksa dan disetujui oleh

| Dosen Pembimbing I | Dosen Pembimbing II |
|---|---|
| Mahendra Data, S.Kom., M.Kom. | Reza Andria Siregar, S.T., M.Kom. |
| NIK: 2015038611171001 | NIP: 19790621 200604 1 003 |

Mengetahui
Ketua Jurusan Teknik Informatika

Tri Astoto Kurniawan, S.T., M.T, Ph.D
NIP. 19710518 200312 1 001

# PERNYATAAN ORISINALITAS

Saya menyatakan dengan sebenar-benarnya bahwa sepanjang pengetahuan saya, di dalam naskah skripsi ini tidak terdapat karya ilmiah yang pernah diajukan oleh orang lain untuk memperoleh gelar akademik di suatu perguruan tinggi, dan tidak terdapat karya atau pendapat yang pernah ditulis atau diterbitkan oleh orang lain, kecuali yang secara tertulis disitasi dalam naskah ini dan disebutkan dalam daftar pustaka.

Apabila ternyata didalam naskah skripsi ini dapat dibuktikan terdapat unsur-unsur plagiasi, saya bersedia skripsi ini digugurkan dan gelar akademik yang telah saya peroleh (sarjana) dibatalkan, serta diproses sesuai dengan peraturan perundang-undangan yang berlaku (UU No. 20 Tahun 2003, Pasal 25 ayat 2 dan Pasal 70).

Malang, 3 Agustus 2018


Yosua Tito Sumbogo

NIM: 145150200111050

# KATA PENGANTAR

Puji syukur penulis panjatkan kehadirat Tuhan Yang Maha Esa karena hanya dengan rahmat dan hidayah-Nya, penulis telah berhasil menyelesaikan Laporan skripsi ini dengan baik. Penulis mengangkat tema ini karena seiring dengan perkembangan teknologi di bidang informatika dan komputer yang mengalami perkembangan pesat, kecepatan, ketepatan, kemudahan serta efisiensi dan efektifitas dianggap sebagai tantangan dalam melakukan suatu kegiatan dan membawa dampak yang signifikan terhadap kehidupan manusia. Penulis mengucapkan banyak terima kasih kepada :

1. Tuhan Yang Maha Esa, yang telah memberikan perlindungan dan kemudahan selama kerja praktik berlangsung maupun dalam penyelesaian laporan praktik kerja lapangan ini.
2. Bapak, Ibu, seluruh keluarga, dan seluruh teman atas segenap dukungan dan kasih sayang yang telah diberikan.
3. Bapak Mahendra Data, S.Kom., M.Kom. dan Bapak Reza Andria Siregar, S.T., M.Kom selaku dosen pembimbing selama pelaksanaan skripsi yang selalu memberikan saran dan masukan kepada penulis
4. Bapak Wayan Firdaus Mahmudi, S.Si., M.T., Ph.D selaku Dekan Fakultas Ilmu Komputer Universitas Brawijaya Malang.
5. Bapak Tri Astoto Kurniawan, S.T., M.T., Ph.D selaku Ketua Jurusan Teknik Informatika Fakultas Ilmu Komputer Universitas Brawijaya Malang.
6. Bapak Agus Wahyu Widodo, S.T., M.CS selaku Ketua Program Studi Teknik Informatika Fakultas Ilmu Komputer Universitas Brawijaya Malang.
7. Seluruh teman-teman yang telah mendukung(Alan, Iskar, Afif, Kevin, Rafael, Devi, Putri, Dhika, David, Maria, Reka, Sherly, Widi, Hermawan) yang selalu mendukung saya, dan membantu saya selama proses pengerjaan skripsi berlangsung serta kawan-kawan saya yang lainnya yang tidak disebutkan.

Dengan segala kerendahan hati, penulis menyadari bahwa laporan ini masih memiliki banyak kekurangan, maka kami mengharapkan kritik dan saran dari para pembaca yang bersifat membangun. Akhirnya, semoga laporan ini dapat bermanfaat dan memberikan informasi bagi kita semua.

Malang, 18 Juli 2018

Penulis,

yosuatito@gmail.com

# ABSTRAK

**Yosua Tito Sumbogo, Implementasi *Failover* Dan *Autoscaling* Kontainer Web Server Nginx Pada Docker Menggunakan Kubernetes**

**Pembimbing: Mahendra Data, S.Kom., M.Kom. dan Reza Andria Siregar, S.T., M.Kom.**

*Web server* merupakan salah satu komponen penting didalam proses *hosting* sebuah *website*. *Web server* yang baik harus mampu melayanai *request* dari pengguna selama 24 jam 7 hari dimana jika *web server* mati maka layanan *website* akan berhenti. Kondisi *web server* tidak dapat bekerja atau *failure* memiki beberapa penyebab seperti putusnya sumber daya listrik, malfungsi perangkat keras, *crash* pada *web server*, dan kegagalan jaringan. NGINX merupakan salah satu *web server* yang populer digunakan, *web server* tersebut mampu di-*deploy* pada komputer *server* secara virtual *menggunakan* metode kontainer dimana virtualisasi hanya dilakukan pada level sistem operasi, hal ini memudahkan proses *deployment* karena kontainer bersifat *lightweight* dan portabel. Agar *web server* mampu terus melayani *request* dari pengguna maka diperlukan mekanisme *failover* dan *autoscaling*. *Failover* mampu melakukan *back up* layanan untuk memastikan *web server* terus bekerja, sementara *autoscaling* mampu membuat layanan *web server* beradaptasi sesuai CPU *usage* yang dibutuhkan untuk menangani *request*. Kubernetes merupakan perangkat lunak yang memiliki mekanisme *failover* dan *autoscaling* kontainer berbasis Docker. Kubernetes mampu memanajemen kontainer berjumlah banyak dan bersifat *open source* sehingga biaya penelitian mampu diminimalkan. Pada penelitian ini jumlah perangkat keras yang digunakan adalah tiga *server* yang membentuk sebuah *cluster* menggunakan Kubernetes, dimana satu *server* akan berperan sebagai *master* dan dua *server* sebagai *minion*. Pengujian yang dilakukan adalah menguji fungsionalitas sistem seperti menjalankan *web server*, menguji mekanisme *failover* dan *autoscaling*, dan non fungsionalitas seperti uji koneksi antar *server* pada *cluster*, rata-rata waktu yang dibutuhkan untuk melaksanakan *failover* dan *autoscaling* serta CPU *usage* yang dihasilkan pada saat *scaling.* Hasil dari penelitian didapat rata-rata waktu yang dibutuhkan untuk *failover* adalah 264,74s untuk *node failure*, dan 3s untuk *service failure*. Untuk rata-rata waktu *autoscaling* sistem membutuhkan 45s untuk *scale up* dan 120s untuk *Scale down*. *Autoscaling* Kubernetes mampu mengurangi CPU *usage* sebesar 0,4% dibanding sistem yang tidak menggunakan *autoscaling*. Sistem yang dibangun mampu menjalankan *web server* secara virtual dengan metode *clustering.*

Kata kunci: Kubernetes, Docker, kontainer, *web server*, NGINX, *failover*, *autoscaling*.

# ABSTRACT

**Yosua Tito Sumbogo, Implementasi *Failover* Dan *Autoscaling* Kontainer Web Server Nginx Pada Docker Menggunakan Kubernetes**

**Pembimbing: Mahendra Data, S.Kom., M.Kom. dan Reza Andria Siregar, S.T., M.Kom.**

Web server is one of the important things in the process of hosting a website. A good web server must be ready to serve requests from users for 24 hours 7 days where if the web server is dead then the website service will stop. The condition of the web server can not work or failure to discern some causes such as electricity resource depletion, hardware crash, crash on the web server, and network failure. NGINX is one web server sought by, that web server is capable of being deployed on a virtual server computer using a method where virtualization is only done at the operating system level, making it easier to process because the container is lightweight and portable. In order for the web server to continue to serve requests from users it will require *failover* and *autoscaling*. *Failover* is able to back up services to ensure the web server continues to work, while autoscalling make a web server service count matches the CPU usage required to request. Kubernetes is software that has a *failover* and *autoscaling* container based on Docker. Kubernetes is able to manage a lot of container and a open source so it will minimized research costs. In this research, the number of devices used are three servers that make a cluster using Kubernetes, where one server will search as master and two servers as minion. The tests performed are functional system functionalities such as running web servers, *failover* and *autoscaling*, and non functionality such as server-to-server in cluster connections, average time required for *failover* and *autoscaling* and CPU usage generated at scaling. The results of the cases found average time required for *failover* were 264,74s for node failure, and 3s for service failure. For an average time-consuming *autoscaling* system it requires 45s to scale and 120s to *Scale down*. Kubernetes automatic scaling can reduce CPU usage by 0,4% compared to systems that do not use *autoscaling*. The built system is able to run the virtual server with clustering method.

*Keywords*: Kubernetes, Docker, kontainer, *web server*, NGINX, *failover*, *autoscaling*.

# DAFTAR ISI

# DAFTAR TABEL

# DAFTAR GAMBAR

# BAB 1 PENDAHULUAN

## 1.1 Latar belakang

*Website* atau situs web merupakan tempat untuk menyebarkan informasi seperti profil perusahaan, hasil penelitian, *e-commerce*, tulisan, gambar, video dan masih banyak lagi secara *online*. Secara sederhana *website* terdiri dari beberapa halaman yang saling terhubung untuk menyalurkan informasi (Yuhefizar, Mooduto, & Hidayat, 2008). Untuk menjalankan sebuah website diperlukan *server* yang berjalan selama 24 jam untuk melayani permintaan dari pengguna. *Web server* merupakan *server* yang berfungsi untuk melayani *request* dan *response* pengguna menggunakan protokol HTTP (*Hyper Text Transfer Protocol*) sebagai protokol pengiriman data (Gourley & Totty, 2002). sebuah website akan disimpan pada *web server* sehingga ketika seorang pengguna me-*request* halaman *website, browser* akan mengirimkan sebuah HTTP *request* ke *web server* lalu *web server* akan mencari *website* tersebut dan membalas dengan *response* halaman *website* yang diminta.

NGINX merupakan salah satu *web server* yang dapat digunakan untuk menjalankan sebuah situs. NGINX bersifat *open source* dan memiliki banyak fitur seperti *load balancer, reverse proxy, IMAP/POP3 proxy server.* NGINX dapat menjalankan situs dengan mekanisme pertukaran data menggunakan protokol HTTP dan memiliki banyak dukungan pengguna (NGINX, 2018).

*Website* menjadi sangat popular digunakan tidak hanya pada situs pendidikan dan pemerintahan, namun juga pada lingkup individu masyarakat karena dengan adanya website memudahkan pengguna untuk bertukar informasi. Sebuah situs website populer seperti facebook, twitter dan google memiliki jumlah kunjungan situs hingga miliyaran pengguna setiap harinya hal ini membuat kinerja dari sebuah *web server* dituntut agar bisa selalu tersedia setiap saat dengan performa yang tinggi sehingga situs selalu dapat diakses.

Masalah yang sering dialami ketika banyak pengunjung mengakses sebuah situs adalah *server* yang berperan sebagai *web server* tidak mampu menangani permintaan sehingga layanan tidak dapat diproses dengan maksimal. Terdapat masalah seperti *failure* atau kegagalan, dimana pada kondisi tersebut *server* tidak dapat melakukan layanan. Beberapa penyebab server failure adalah putusnya sumber daya listrik, malfungsi perangkat keras, sistem operasi yang *crash* dan kegagalan jaringan (oracle, 2018). Salah satu cara untuk mengatasi *failure* adalah melakukan mekanisme *failover* yaitu sebuah proses pergantian layanan dalam hal ini *web server* dari *server* yang mengalami *failure* menuju *server* lain yang dalam keadaan *standby*, mekanisme *failover* berjalan secara otomatis (Jayaswal, 2005).

Permasalahan lain yang dapat terjadi adalah terkait kemampuan server dalam menangani permintaan dari pengguna seperti menerima *request* dan mengirim

*response* HTTP, kemampuan sebuah server tentu berbeda jika dikerjakan oleh dua atau lebih server dengan kemampuan yang sama. Proses penambahan layanan atau *server* tanpa mengganggu kinerja *server* disebut *scalability* atau skalabilitas (Bondi, 2000). Terdapat pula sebuah metode *autoscaling* dimana sistem akan memonitoring sebuah *server cluster* secara berkala, ketika sistem mendeteksi layanan pada sebuah server tidak dapat memenuhi permintaan pengguna maka sistem akan menambahkan layanan tersebut ke *server* lain yang tersedia (Armbrust, Fox, Griffith, Joseph, & Katz, 2009).

Untuk dapat mengatasi masalah *failure* dan skalabilitas dapat menggunakan sebuah metode *server cluster* yaitu, membangun sistem yang terdiri lebih dari beberapa server dan bekerja sama untuk menjalankan sebuah layanan seperti *web server* (Oracle, 2018). Pada *server cluster* untuk menghemat *resource* dapat digunakan sebuah teknik virtualisasi yang memanfaatkan kontainer. Kontainer sendiri adalah sebuah wadah terisolasi yang berisi aplikasi beserta dependency, dan, library yang dibutuhkan untuk aplikasi tersebut berjalan (Mouat, 2015). Kontainer akan berjalan diatas lapisan sistem operasi dan berbagi kernel dengan sistem operasi host sehingga bersifat lightweight, dan portable. Layanan seperti *web server* NGINX dapat dimuat kedalam kontainer lalu didistribusikan kepada *server cluster*. Docker merupakan platform yang menyediakan virtualisasi berupa kontainer. Docker menawarkan fitur seperti *portability, agility, security, cost savings* (Docker, 2017).

Untuk menjalankan mekanisme *failover* dan *autoscaling* pada *server cluster* diperlukan sebuah kontainer *orchestration* yang berfungsi untuk mengatur dan memonitoring kontainer pada *cluster*. Dimana sistem dapat melihat statistik keadaan *server* sehingga secara otomatis dapat menjadwalkan layanan seperti *web server* dalam bentuk kontainer kedalam *cluster.*

Kubernetes merupakan sebuah kontainer *orchestration* yang memiliki fungsi untuk memonitoring, men-*deploy, autoscaling* kontainer pada sebuah *server cluster* (Kubernetes, What is Kubernetes, 2017). Kubernetes memiliki fitur yang tidak dimiliki kontainer *orchestration* lain seperti fitur *autoscaling* yaitu Kubernetes akan secara berkala mengecek *cpu* dari masing-masing node dimana jika *cpu* pada sebuah node telah mencapai batas parameter yang telah ditentukan maka Kubernetes akan mereplikasi layanan ke *node* lain yang tersedia. Kubernetes mendukung *platform* kontainer sepert Docker dan bersifat *open source* sehingga mudah dan murah untuk dikembangkan (Kubernetes, What is Kubernetes, 2017).

Penelitian ini akan berfokus kepada bagaimana cara merancang dan mengimplementasikan sistem *autoscaling* dan *failover* kontainer pada *platform* Docker menggunakan kubernetes pada *server cluster* yang dijalankan secara virtual dan menjalankan layanan *web server* NGINX. Penelitian ini akan dilakukan pada lingkungan terisolasi yang terkontrol dan secara virtual.

## 1.2 Rumusan masalah

Berdasarkan latar belakang yang telah dijelaskan dapat ditentukan beberapa rumusan masalah sebagai berikut:

1. Bagaimana merancang dan mengimplementasikan *web server* berbasis kontainer dengan menggunakan Kubernetes?.
2. Bagaimana merancang dan mengimplementasikan mekanisme *failover* dan *autoscaling* pada *web server* berbasis kontainer dengan menggunakan Kubernetes?.
3. Bagaimana hasil *failover* dan *autoscaling* layanan *web server* NGINX berbasis kontainer menggunakan kubernetes?.

## 1.3 Tujuan

Tujuan yang diharapkan dari penelitian ini adalah mengetahui bagaimana cara merancang dan mengimplementasikan sebuah infrastruktur *server* berbasis kontainer yang dapat menyediakan layanan *web server* sehingga mampu menghadapai masalah skalabilitas dan *failure* pada *web server*.

## 1.4 Manfaat

Manfaat dari penelitian ini adalah sebagai berikut

1. Bagi peneliti
   - Menjadi media untuk mengimplementasikan ilmu yang didapat selama perkuliahan.
   - Dapat mempelajari lebih dalam metode kontainer dan *cloud.*
   - Dapat digunakan sebagai referensi penelitian yang lain.
2. Bagi pengguna
   - Dapat menjadi sebuah solusi perancangan untuk mengatasi permasalahan skalabilitas dan *failure* pada *web server*.
   - Dapat menjadi referensi arsitektur sistem *server* berbasis kontainer.

## 1.5 Batasan masalah

Penelitian yang akan dilakukan terbatas pada:

1. Penelitian akan dilakukan secara virtual dikarenakan sistem yang dibuat berupa prototipe dan memiliki resiko mengganggu kinerja *server* sebenarnya.
2. Penelitian hanya akan berfokus pada sistem *failover* dan *autoscaling* kontainer menggunakan Docker dan kontainer *orchestration* Kubernetes.
3. Layanan yang dijalankan hanya *web server* NGINX sesuai permasalahan yang telah dijelaskan pada latar belakang.

## 1.6 Sistematika pembahasan

Sistematika pembahasan pada penelitian yang akan dilakukan adalah sebagaia berikut:

**BAB 1 PENDAHULUAN**

Pada bab ini berisi mengenai latar belakang masalah yang menjelaskan permasalahan yang terjadi beserta solusi yang ditawarkan, rumusan masalah dalam penelitian, tujuan penelitian, manfaat yang didapat dari penelitian, Batasan masalah penelitian, dan sistematika penulisan laporan penelitian.

**BAB 2 LANDASAN KEPUSTAKAAN**

Pada bab ini membahas mengenai penelitian sebelumnya yang berkaitan dengan scaling, dan virtualisasi menggunakan kontainer serta beberapa teori dan penjelasan dari referensi yang menjadi dasar keilmuan pada penelitian.

**BAB 3 METODOLOGI**

Pada bab ini menjelsankan mengenai metodologi penelitan yang akan diterapkan seperti studi literature, analisis kebutuhan, perancangan sistem, implementasi sistem, pengujian dan analisis, serta kesimpulan. bab ini juga berisi analisis kebutuhan dari sistem yang akan dibangun baik kebutuhan fungsional dan non fungsional, serta perancangan dari sistem baik perancangan perangkat lunak, perangkat keras, implementasi, dan pengujian sistem.

**BAB 4 PERANCANGAN DAN IMPLEMENTASI**

Pada bab ini berisi bagaimana proses perancangan dan implementasi pada sistem, dimulai dari perancangan lingkukan kerja, source code yang digunakan serta komponen-komponen pendukung yang digunakan pada sistem.

**BAB 5 PENGUJIAN DAN ANALISIS HASIL**

Bab ini berisi proses pengujian yang dilakukan pada sistem sesuai dengan skenario yang telah dirancang pada bab perancangan. Hasil dari pengujian kemudian akan dianalisis dan ditentukan kesimpulan dari pengujian tersebut.

**BAB 6 PENUTUP**

Pada bab ini akan berisi mengenai kesimpulan dari sistem yang telah dibangun apakah solusi yang ditawarkan mampu menyelesaikan masalah yang ada atau tidak, serta saran untuk penelitian berikutnya.

# BAB 2 LANDASAN KEPUSTAKAAN

## 2.1 Kajian pustaka

Terdapat beberapa penelitian sebelumnya terkait Kubernetes seperti yang dilakukan oleh Víctor Medel yaitu bagaimana kinerja dan manajemen sumber daya yang dilakukan oleh Kubernetes. Penelitian yang dilakukan menghasilkan sebuah model manajemen sumber daya baru bernama *net-based model.* Penelitian ini juga menyatakan bahwa penggunaan Kubernetes sebagai kontainer *orchestration* mempermudah arsitektur sistem *cloud native application* karena mendukung skalabilitas dan *deployment* aplikasi yang cepat. (Medel, Rana, Bañares, & Arronategui, 2016). Terdapat juga sebuah penelitian yang dilakukan Dwiyanto Gunawan mengenai Rancang Bangun Arsitektur Laboratorium Virtual Jaringan Komputer Dengan Virtualisasi Berbasis Kontainer pada penelitian ini menggunakan Docker sebagai platform virtualisasi kontainer. pemanfaatan kontainer digunakan sebagai komputer virtual sementara untuk mengatur sumber daya perangkat keras digunakan Openstack. Namun pada penelitian ini belum menunjukan bagaimana cara mengatur jumlah kontainer yang banyak. Sistem yang dibangun pun terbatas pada bagian *back end* sistem. hasil dari penelitian tersebut komputer virtual yang dibangun menggunakan kontainer mampu dibuat dalam waktu kurang dari tiga detik dan jumlah komputer yang dipesan sangat mempengaruhi proses pembuatan komputer virtual (Gunawan, Basuki, & Bhawiyuga, 2016). Pada penelitian lain terkait skalabilitas terdapat sebuah fitur *autoscaling* yang memanfaatkan metode prediksi *Simple Moving Average.* Metode ini memanfaatkan data dari masa lalu untuk memprediksi data masa depan, data yang dimanfaatkan adalah *time series* yaitu proses penetuan *pattern* dari riwayat kejadian suatu data (Suwastika, W, & Harsono, 2015). Namun metode *autoscaling* yang digunakan diimplementasikan pada perangkat lunak OpenStack bukan kontainer.

## 2.2 Virtualisasi

Definisi virtual menurut Kamus Besar Bahasa Indonesia adalah secara nyata, sedangakan akhiran -isasi memiliki makna sebuah proses, usaha, atau metode. Secara sederhana Virtualisasi dapat diartikan sebuah proses untuk membuat sesuatu menjadi nyata.

Virtualisasi merupakan sebuah metode dalam mengabstrakkan aplikasi, dan komponen perangkat keras dalam bentuk logika atau virtual. Tujuan dari virtualisasi adalah skalabilitas, *availability,* membuat domain manajemen dan sistem keamanan yang terpadu (Kusnetzky, 2011).

Virtualisasi pada perangkat keras dapat diartikan sebagai upaya mengabstrabskkan perangkat keras beserta sistem operasinya menjadi bentuk logika. Teknologi virtual lebih sering dimanfaatkan sebagai upaya mensimulasikan sebuah lingkungan kerja dari sistem operasi sehingga pengguna tidak perlu membeli perangkat keras baru untuk memiliki sistem operasi yang berbeda.

Dengan memanfaatkan teknologi virtual pengguna dapat membuat sebuah sistem secara terpisah seperti *Storage Area Network* dengan layanan *Virtual LAN* sehingga masing-masing layanan memiliki tingkat *availability* yang tinggi dan tidak terikat satu dengan yang lain (Portnoy, 2016).

Pengguna dapat memanfaatkan sumber daya yang sedang tidak terpakai seperti CPU, RAM, jaringan komputer menjadi beberapa mesin virtual untuk menjalankan beberapa pekerjaan sehingga sumber daya dapat digunakan secara maksimal. Pada dasarnya terdapat empat teknik untuk melakukan virtualisasi yaitu *Emulation, Hypervisor, Full virtualization, Para virtualization* (Durairaj & Kannan, 2014).

### *2.5.1 Emulation*

*Emulation* merupakan teknik virtualisasi yang mengubah perilaku dari perangkat keras menjadi program komputer dan terletak pada sistem operasi. *Emulation* memiliki tingkat fleksibilitas yang tinggi pada sistem operasi namun memiliki tingkat translasi yang rendah bila dibandingkan dengan *Hypervisor*. *Emulation* membutuhkan konfigurasi perangkat keras yang baik untuk menjalankan perangkat lunak (Durairaj & Kannan, 2014).

### *2.5.2 Hypervisor*

*Hypervisor* merupakan teknik virtualisasi yang berjalan diantara lapisan perangkat lunak dan perangkat keras untuk memonitor dan memvirtualkan sumberdaya dari komputer *host* sesuai dengan keinginan pengguna. Hypervisor diklasifikasikan sebagai *native* dan *hosted* dimana *native based* akan berjalan pada perangkat keras sedangkan *host based* akan berjalan pada sistem operasi *host* dan membuat sumber daya virtual seperti *CPU, Memory*, *storage,* dan *drivers* (Durairaj & Kannan, 2014).

### *2.5.3 Full virtualization*

Pada teknik *Full virtualization Hypervisor* akan membuat sebuah lingkungan terisolasi antara *guest* atau *server* virtual dengan perangkat keras *host* atau *server*. *Full virtualization* akan mensimulasikan semua perangkat keras kedalam perangkat lunak sehingga dapat bekerja seperti perangkat keras sebenarnya (Durairaj & Kannan, 2014).

### *2.5.4 Para virtualization*

Pada teknik ini virtualisasi hanya dilakukan ditingkat modifikasi atau kompilasi dari *guest* OS lalu dijalankan seperti proses biasa tanpa perlu mengakses perangkat lunak. Pada perangkat keras virtualisasi hanya bekerja untuk penjadwalan sumber daya dan *multiplexing* (Durairaj & Kannan, 2014).

## 2.3 Kontainer

Kontainer merupakan sebuah teknik untuk mengisolasi aplikasi dengan *libraries* dan *dependency* dari aplikasi tersebut. Dengan menggunakan kontainer pengguna dapat menjalankan aplikasi secara terisolir dari lingkungan *host*. Kontainer bersifat *lightweight* karena *host* akan berbagi *kernel* dengan kontainer

sehingga tidak perlu untuk meng-*install* sistem operasi didalam kontainer tersebut (Mouat, 2015).

Kontainer memiliki beberapa kelebihan yaitu:

1. Kontainer akan berbagi sumber daya dengan *host* OS sehingga lebih efisien, hal ini membuat kontainer memiliki waktu mulai dan berhenti yang cepat.
2. Tingkat portabilitas kontainer sangat tinggi karena berbagi kernel dengan *host* sehingga dapat berjalan disemua lingkungan sistem operasi yang mendukung sistem kontainer dan dapat menyelesaikan masalah depedensi.
3. Pengguna dapat menjalankan banyak proses pada kontainer yang berbeda pada saat yang bersamaan karena kontainer bersifat *lightweight.*
4. Proses *deployment* pada menjadi lebih mudah karena pengguna lain tidak perlu mengkonfigurasi aplikasi dari awal karena semua kebutuhan telah disediakan pada *images* kontainer. (Mouat, 2015).

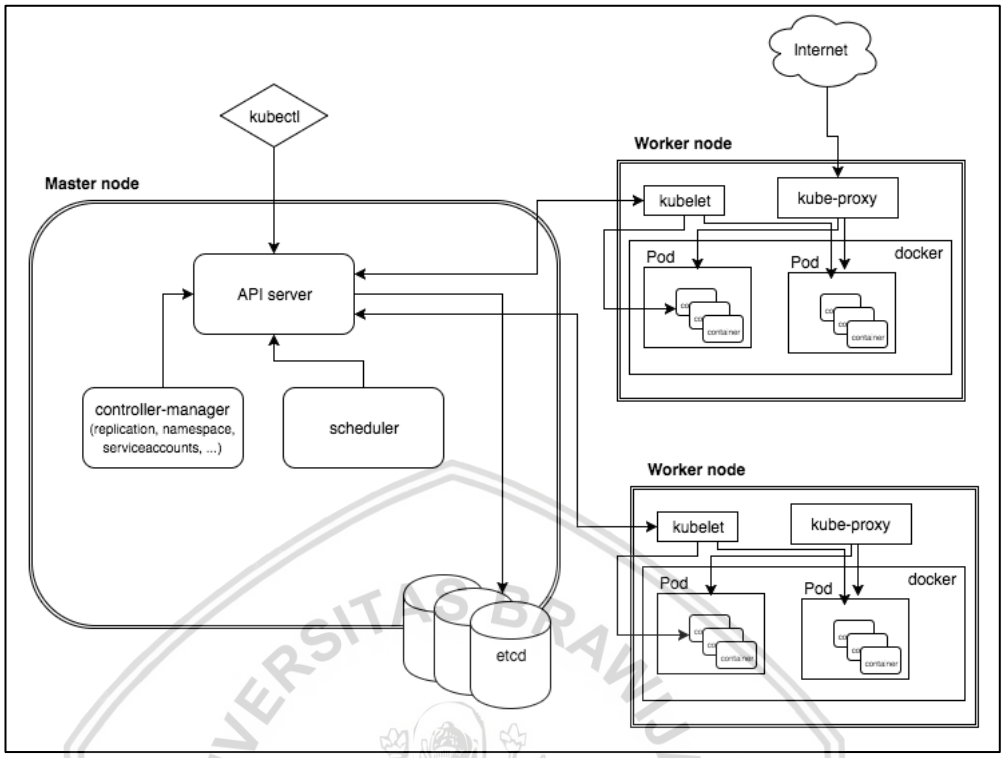
Namun kontainer juga memiliki beberapa kelemahan karena berbagi *kernel* dengan sistem operasi *host* maka kontainer juga memiliki hak akses *root* sehingga rentan dalam serangan seperti perangkat lunak berbahaya dapat menyerang langsun sistem operasi *host* dan kontainer *yang lain*, selain itu kontainer tidak dapat berjalan di sistem operasi yang berbeda seperti kontainer Docker pada Linux tidak dapat berjalan pada Windows *server* (Bigelow, 2015).

## 2.4 Docker

Docker merupakan platform *open source* bersifat *lightweight* yang memudahkan pengguna seperti sistem admin didalam proses *deployment* sebuah aplikasi berbasis kontainer (Turnbull, 2015). Docker dapat berjalan pada *host* yang paling minimal seperti ubuntu atau CentOS *server*, dengan menggunakan Docker pengembang tidak perlu memusingkan depedensi dan lingkungan kerja karena aplikasi berada didalam kontainer sementara tim *operations* hanya perlu mengurusi masalah manajemen kontainer (Turnbull, 2015).

## 2.5 Kubernetes

Kubernetes merupakan *platform open source* yang berfungsi sebagai kontainer *orchestration* yaitu *platform* yang akan bertugas melakukan penjadwalan, *scaling*, *recovery* dan monitoring pada kontainer (Medel, Rana, Bañares, & Arronategui, 2016). Kubernetes menggunakan sistem *cluster-based* dimana sekumpulan *node* seperti server akan membentuk *cluster* dan dikontrol oleh sebuah *master node*, Kubernetes akan memastikan setiap kontainer berjalan sesuai keingingan pengguna seperti menjaga jumlah kontainer yang dijalankan, menjadwalkan pekerjaan kontainer pada *host* didalam *cluster,* lalu men-*scaling* kontainer sesuai kebutuhan (Medel, Rana, Bañares, & Arronategui, 2016). Gambar 2.1 menunjukan arsitektur dari Kubernetes.




**Gambar 2.1 Arsitektur Kubernetes**

Sumber: (Isabekyan, 2016)

Pada Gambar 2.1 dapat dilihat terdapat banyak komponen dari *node master* dan *worker* berikut penjelasan dari masing-masing komponen:

***Node Master***

*Node master* berfungsi sebagai *node* pengontrol pada *cluster, node* akan mengatur dan menjadwalkan pekerjaan pada masing-masing *node worker*.

***API server***

*API server* berfungsi sebagai pintu masuk perintah sistem yang menggunakan protokol *REST, API server* akan menerima permintaan, memvalidasi dan mengeksekusi permintaan.

***etcd storage***

*etcd storage* merupakan tempat penyimpanan dan pendistribusian konfigurasi dari *cluster* seperti *token, namespaces,* pekerjaan yang telah dijadwalkan, dan masih banyak lagi.

***scheduler***

*scheduler* berfungsi untuk menjadwalkan pekerjaan pada *node*, *scheduler* memiliki informasi mengenai sumber daya yang dimilik *node* pada *cluster.*

***controller-manager***

Merupakan komponen yang mengontrol semua pekerjaan pada *cluster* seperti memeriksa jumlah pod apakah sesuai dengan persyaratan yang telah ditentukan pengguna atau tidak.

***Node Worker***

*Node Worker* merupakan tempat dari layanan didalam *cluster* berjalan, *node worker* akan berhubungan dengan *node master* melalu *API server.*

***Docker***

*Docker* merupakan platform yang didukung oleh kubernetes dan akan berjalan disetiap *node worker*. *Docker* bertugas untuk mengunduh *images* dan menjalankan kontainer

***kubelet***

*kubelet* akan mendapatkan konfigurasi proses dari *node master* dan memastikan kontainer berjalan sesuai konfigurasi. *kubelet* juga memiliki fungsi untuk berkomunikasi dengan *node master* dan etcd untuk mendapatkan informasi mengenai konfigurasi dari proses.

***kube-proxy***

Berperan sebagai proxy jaringan dan *load balancer* untuk layanan pada sebuah *node*. *kube-proxy* akan bertugas untuk mengurus paket TCP dan UDP

***kubectl***

Berfungsi sebagai perintah berbentuk CLI atau *command line interface* untuk berkomunikasi dengan API dan mengirim perintah kepada *node master*. sumber: (Isabekyan, 2016)

Terdapat beberapa istilah pada kubernetes yang harus dipahami seperti Pod, Pod adalah kumpulan dari beberapa kontainer yang membentuk *microservices* dan berkomunikasi satu dengan lainnya dan merupakan unit terkecil pada kubernetes (Isabekyan, 2016). Pod bersifat sementara, diciptakan dan dihancurkan sesuai kebutuhan pada sistem. agar memudahkan pod berkomunikasi dengan lingkungan diluar *cluster* maka dibutuhkan sebuah *Virtual IP* fungsi ini disebut *service* (Isabekyan, 2016).

## 2.6 *Failover*

*Failover* merupakan sebuah mekanisme pergantian sebuah layanan seperti *software* atau *hardware* dari sebuah *node* yang sedang mengalami *failure* ke *node* lain yang tersedia didalam sistem. Mekanisme ini dapat berjalan secara otomatis dan menggantikan layanan yang mengalami *failure* seperti sebelumnya (Jayaswal, 2005).

## 2.7 *Autoscaling*

*Scaling* merupakan kemampuan sistem untuk menyesuaikan sumber daya yang dimiliki seperti menurunkan atau menambah jumlah proses sesuai kebutuhan sistem tanpa mengganggu proses yang sedang berjalan (Bhowmik,

2017). fitur *scaling* pada virtualisasi sangat dibutuhkan demi menjamin sifat *availability* dimana layanan harus tetap berjalan walaupun sedang mengalami penambahan atau pengurangan sumber daya. Kubernetes memiliki fitusr *autoscaling* dimana sistem dapat mengatur sumber daya yang ada seperti *microservice* sesuai kebutuhan pengguna secara otomatis sesuai parameter sumber daya yang telah diatur. Namun saat ini parameter yang dapat dimanfaatkan hanya CPU dari masing-masing *server* (Kubernetes, What is Kubernetes, 2017).

## 2.8 *Web server*

*Web server* merupakan sebuah perangkat lunak yang didesain untuk menerima permintaan dari pengguna dan membalasnya dengan mengirimkan halaman web berupa HTML atau *Hypertext Markup Language* menggunakan protokol HTTP atau HTTPS *(Hypertext Transfer Protocol Secure)* sebagai protokol pertukaran data (Solichin, 2016). *Web server* juga dapat diartikan sebagai perangkat keras tempat menyimpan dan mempublikasikan situs secara *online* (webdevelopersnotes, 2018). Banyak *web server* yang populer digunakan seperti Apache, dan NGINX

## 2.9 NGINX

NGINX merupakan salah satu *web server* yang dikembangkan oleh Igor Sysoev pada tahun 2002. NGINX merupakan *web server* yang bekerja menggunakan arsitektur sistem asinkronus, *non-blocking, event-driven connection* dimana secara berkala setiap proses terjadi jika ada *event* seperti koneksi yang masuk seperti permintaan dari pengguna dan balasan dari *server* (NGINX, 2018). Karena bekerja secara asinkronus maka NGINX dapat menghandle koneksi yang banyak dimana pada awal perkembangan NGINX digunakan untuk menyelesaikan masalah C10K atau 10.000 koneksi pada satu waktu.

## 2.10 Kubespray

Kubespray merupakan proyek *opensource* yang dimiliki oleh komunitas penggiat Kubernetes dan berfungsi sebagai cara *deployment* Kubernetes kedalam *server cluster* menggunakan bantuan Ansible (Kubespray, 2018).

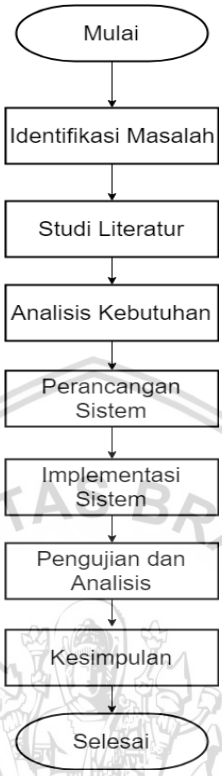
## 2.11 Ansible

Ansible merupakan *platform* automasi sebuah sistem atau aplikasi yang memudahkan pengembang untuk melakukan hal-hal *repetitive* seperti mengupdate sistem atau mendeploy aplikasi kedalam sistem secara otomatis (Ansible, 2018).

## 2.12 Linux KVM

KVM(*Kernel-based Virtual Machine*) merupakan metode virtualisasi yang bersifat *full virtualization* dimana KVM menyediakan infrastruktur virtualisasi dan berperan sebagai *hypervisor.* KVM berguna untuk menjalankan beberapa *images* virtual seperti Linux atau Windows, setiap mesin virtual yang dijalankan memiliki beberapa perangkat keras virtual seperti kartu jaringan, penyimapanan, RAM, prosesor (linux-kvm, 2018).

# BAB 3 METODOLOGI




**Gambar 3.1 Metodologi penelitian**

Bab ini menjelaskan metodologi penelitian yang dilakukan secara rinci. Pada Gambar 3.1 merupakan diagram alir dari metodologi penelitian yang akan dilaksanakan.

## 3.1 Identifikasi masalah

Pada tahapan ini adalah penetuan masalah yang akan diteliti solusinya. Sesuai latar belakang yang telah dipaparkan peneliti akan melakukan perancangan dan pengimplementasian sistem *failover* dan *autoscaling* kontainer pada Docker. Studi kasus yang diangkat adalah *failure* dan skalabilitas pada *web server* NGINX dimana sistem harus dapat melakukan *failover* saat sebuah layanan atau *server* sedang down dan sistem juga dapat melakukan *scaling* pada layanan ketika permintaan dari pengguna tinggi sehingga layanan tetap dapat diakses.

## 3.2 Studi literatur

Studi literatur merupakan cara yang ditempuh untuk menambah referensi dan pengetahuan didalam proses penelitian. Studi literatur diperlukan sebagai dasar teori dan mempelajari penelitian terkait yang pernah dilakukan oleh peneliti lain. Berikut studi literatur yang akan dipelajari:

1. **Virtualisasi**

Mempelajari konsep dan jenis-jenis teknik virtualisasi yang ada serta bagaimana cara mengimplementasi kedalam sistem.

2. **Kontainer**
   Mempelajari konsep dari Kontainer sebagai salah satu teknik virtulasisasi.
3. **Kubespray**
   Mempelajarai cara *deployment* Kubernetes menggunakan Kubespray.
4. **Docker**
   Mempelajari Docker sebagai platform penyedia kontainer sebagai proses implementasi.
5. **Kubernetes**
   Mempelajari konsep, cara penggunaan, dan implentasi Kubernetes sebagai *container orchestration.*
6. ***Scaling***
   Mempelajari konsep *scaling* dan *autoscaling* pada Kubernetes.
7. ***Web server***
   Mempelajari konsep dan cara penggunaan dari *web server.*
8. **NGINX**
   Mempelajari *software* NGINX yang berperan sebagai *web server.*
9. **Linux KVM**
   Mempelajari bagaimana cara membuat lingkungan virtual didalam proses penelitian.
10. **Ansible**
    Mempelajari Ansible sebagai alat bantu dalam pengoperasian Kubespray.

## 3.3 Gambaran umum sistem

Sistem yang dibangun dapat menjalankan layanan *web server* NGINX, *web server* akan dijalankan pada sebuah *server cluster* yang menggunakan metode virtualisasi kontainer. Pada *server cluster* terdapat 3 buah komputer virtual yang dijalankan menggunakan Linux KVM dimana komputer virtual akan menjadi *server.* 1 buah komputer virtual berperan sebagai *master* dan minon dan mengontrol 2 buah *minion* lain sehingga sistem dapat dikontrol secara terpusat. Sistem yang dibangun mampu menjaga layanan *web server* tetap tersedia selama 24 jam dalam 7 hari dimana sistem mampu melakukan mekanisme *failover* ketika layanan atau node mengalami *failure*, dan *autoscaling web server* berdasarkan parameter CPU *usage* dari node. Untuk menjalankan mekanisme tersebut maka perangkat lunak Kubernetes akan digunakan. Untuk mempermudah proses *deployment* Kubernetes maka Kubespray akan digunakan untuk menkonfigurasi semua kebutuhan sistem seperti konfigurasi *node*, *tunneling,* instalasi perangkat lunak Docker dan mengatur versi dari perangkat lunak.

## 3.4 Analisis kebutuhan dan perancangan

Pada bagian ini akan dianalisis mengenai kebutuhan apa saja yang dibutuhkan selama proses penelitian. Penelitian yang dilakukan akan bertipe Implementatif dengan jenis perancangan. Perancangan sistem yang akan diimplementasikan pada sebuah lingkungan virtual menggunakan bantuan Linux KVM. Pada penelitian

ini hanya terbatas pada simulasi karena hanya menguji fitur *failover* dan *autoscaling* kontainer pada Docker menggunakan Kubernetes. simulasi dilakukan untuk membuat purwarupa dari sistem untuk diimplementasikan pada *server* sebenarnya. Berdasar pada rumusan masalah yang telah dijelaskan maka kebutuhan yang diperlukan selama proses penelitian adalah:

### 3.4.1 Kebutuhan fungsional

Kebutuhan fungsional merupakan kebutuhan yang dapat dijalankan oleh sistem yang akan dibangun. Berikut kebutuhan fungsional yang diperlukan:

1. Sistem mampu memberikan layanan *web server* secara *clustering.*
2. Sistem mampu menjalankan mekanisme *failover* dan *autoscaling* terhadap layanan *web server.*

### 3.4.2 Kebutuhan non fungsional

1. *Server* dapat berkomunikasi didalam *cluster.*
2. Sistem mampu menyediakan layanan *web server* NGINX.
3. Sistem mampu menyediakan layanan *web server* bersifat *high availability* dengan fitur *failover.*
4. Sistem mampu mengurangi CPU *usage* dari *server* dengan fitur *autoscaling.*

### 3.4.3 Kebutuhan perangkat lunak

- Ansible
- Docker
- Kubespray
- Kubernetes
- NGINX
- Linux KVM
- Putty untuk melakukan *remote access*
- Sistem Operasi Linux

### 3.4.4 Kebutuhan perangkat keras

Komputer *server* yang mampu menjalankan Kubernetes agar layanan *web server* dapat berjalan secara *cluster.*

### 3.4.5 Batasan perancangan

Sistem yang akan dibangun memiliki batasan pada perancangan sebagai berikut:

1. Sistem yang dibangun akan dijalankan pada lingkungan virtual menggunakan Linux KVM.
2. Didalam *server cluster* setiap *node* akan berhubungan dan bertukar data menggunakan *overlay network* yang telah disediakan oleh Kubernetes.
3. Sistem akan terdiri dari tiga node dimana satu node berperan sebagai *master* dan minion, dan dua node sebagai *minion.*
4. Layanan yang dijalankan hanya *web server* NGINX. Layanan tersebut akan meng-*hosting homepage* dari NGINX untuk proses pengujian.

5. Perancangan sistem hanya terbatas untuk menguji mekanisme *failover* dan *autoscaling*.

### 3.4.6 Daftar istilah

Tabel 3.1 merupakan daftar istilah pada sistem yang digunakan pada tahap perancangan beserta penjelasan.

**Tabel 3.1 Daftar istilah**

| **Istilah** | **Penjelasan** |
|---|---|
| *Node* | Perangkat komputer. |
| *Master* | *Node* yang berperan sebagai *controller* pada *cluster server.* |
| *Minion* | *Node* yang berperan sebagai pekerja dan menjalankan layanan pada *cluster server.* |
| *Server* | *Node* yang dapat berperan sebagai *master* atau *minion* yang berfungsi menjalankan layanan pada sistem. |
| *Cluster* | Sekumpulan *server* yang membentuk kelompok untuk menjalankan layanan. |
| *Failover* | Mekanisme *backup* pada layanan ketika sebuah layanan atau node mengalami *failure.* |
| *Autoscaling* | Mekanisme *scaling* secara otomatis pada layanan ketika sebuah layanan atau node tidak mampu melayani permintaan yang ada. |
| kontainer | Teknik virtualisasi pada *layer* sistem operasi yang berbagi kernel dengan sistem operasi dari *node.* |
| *Overlay Network* | *Overlay network* adalah sebuah jaringan yang dibangun diatas jaringan lain pada protokol TCP/IP (Tarkoma, 2010) |

## 3.5 Implementasi sistem

Pada tahap ini hasil dari rancangan akan diimplementasikan pada lingkungan virtual agar lebih efisien dan tidak perlu mengganggu kinerja server sebenarnya. Instalasi dan konfigurasi akan dilakukan berdasarkan referensi dokumentasi dari masing-masing tools yang digunakan.

## 3.6 Pengujian dan analisis

Pada bab ini hasil dari implementasi sistem akan diuji sesuai skenario pengujian yang telah ditentukan dan dianalisis. Pengujian dan analisis hanya terbatas pada mekanisme *failover* dan *autoscaling*. Parameter pengujian yang akan dilakukan adalah menguji *down time* dan *response time* dari layanan saat melakukan *failover* dan *autoscaling*, pengujian ping untuk mengetahui apakah benar layanan telah aktif saat dilakukan *failover* dan *autoscaling,* pengujian *stress* kepada *web server* dengan tujuan menguji kemampuan *autoscaling* dimana CPU digunakan sebgai parameter *autoscaling*.

## 3.7 Kesimpulan

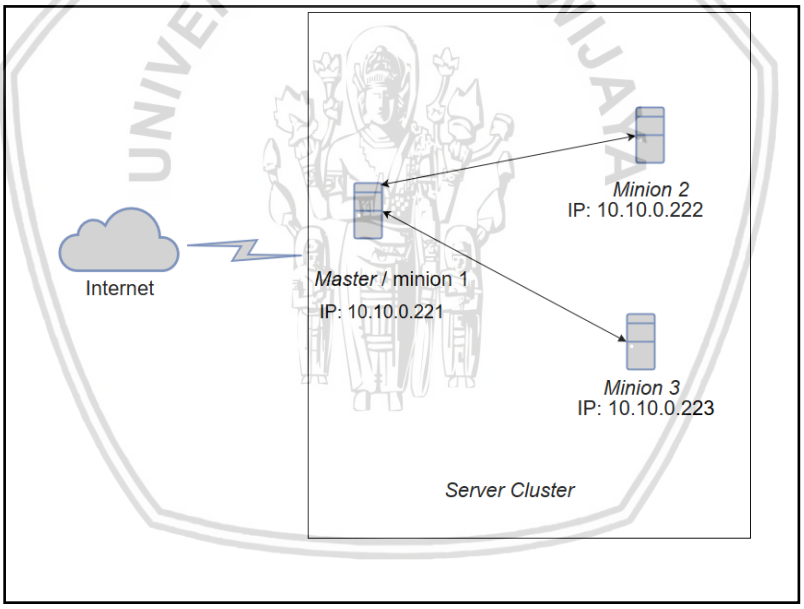
Pada tahap ini akan dilakukan penarikan kesimpulan dan pemberian saran setelah mendapatkan hasil dari pengujian dan analisis sistem. Hasil dari pengujian dan analisis sistem dapat menjadi jawaban dari masalah yang telah dipaparkan serta data dari penelitian yang didapat diharapkan mampu berguna untuk diimplementasikan pada *server* sebenarnya dan dapat menjadi referensi bagi penelitian berikutnya.

# BAB 4 PERANCANGAN DAN IMPLEMENTASI

## 4.1 Perancangan sistem

### 4.1.1 Perancangan *topology* jaringan

Perancangan *topology* jaringan digunakan untuk membuat lingkungan implementasi dan pengujian pada sistem. Pada perancangan yang dibuat semua *node* akan berada pada sebuah *subnet* yang sama. Pada *cluster* akan terdapat 3 buah node yang berperan sebagai 1 *master/minion* dan 2 *minion.* Pada *cluster, node master* akan menjadwalkan dan mengatur layanan *web server* pada masing-masing *minion* sesuai kebutuhan user. Untuk mengakses layanan Kubernetes telah menyediakan sebuah fitur berupa *NodePort* dimana Kubernetes akan mengalokasikan *port number* antara 30000-32767 sehingga layanan dapat diakses dengan cara *IPNode:PortNode.* IP yang akan dimanfaatkan adalah IP dari *node master* walaupun dapat diakses dari IP *node* lain didalam *cluster* dengan *port* yang sama. Data yang dikirim dari pengguna akan di-*forward* oleh *NodePort* menuju *node* yang memiliki layanan *web server*.




**Gambar 4.1 *Topology* jaringan**

Pada gambar 4.1 Menjelaskan tentang *topology* jaringan dari sistem yang akan dibangun. *Topology* yang dipakai adalah *topology* star dimana setiap *node minion* akan terhubung kepada *node master.* Setiap node akan terhubung dengan Tunneling IP Calico yang bertujuan untuk menghubungkan node didalam *cluster*. Pada *topology node master* akan berperan sebagai *minion* untuk mengehemat sumber daya yang dimiliki.

## 4.2 Perancangan *failover*

Pada mekanisme *failover* Kubernetes telah menyediakan fitur untuk mengatasi masalah *failure* yaitu *Deployment*. Fitur ini memastikan bahwa service yang berjalan akan selalu *available.* Kubernetes akan secara periodik memeriksa layanan *web server* dan *node,* apakah mengalami *failure* atau tidak. Layanan *web server* pada percobaan yang dijalankan hanya satu dimana *deployment* akan dikonfigurasi untuk memastikan apakah tersedia satu *web server* pada *node minion*, jika *web server* mengalami *failure* maka Kubernetes akan memeriksa *node* yang ada apakah mampu menjalankan layanan atau tidak jika bisa maka *node* tersebut akan dipilih.

## 4.3 Perancangan *autoscaling*

Pada mekanisme *autoscaling* parameter yang digunakan agar Kubernetes melakukan *scale up* atau *Scale down* adalah CPU dimana masing-masing kontainer *web server* akan diberikan CPU sejumlah tertentu, lalu diberi Batasan dalam persen sehingga ketika CPU load mencapai parameter tertentu Kubernetes akan melakukan *scale up* atau *Scale down* secara otomatis.

Pada proses *scaling,* Kubernetes akan memeriksa keadaan masing-masing *node* apakah *node* yang berada didalam *cluster* mampu menjalankan layanan sesuai permintaan jika tersedia *node* yang memiliki sumber daya yang sesuai maka *node* tersebut akan dipilih untuk menjalankan layanan.

## 4.4 Perancangan pengujian

Pengujian yang dilakukan pada sistem bertujuan untuk mengetahui apakah sistem yang dibangun dapat mengatasi masalah yang ada. Tabel 4.1 akan menjelaskan daftar skenario uji pada kebutuhan fungsional, lalu tabel 4.2 akan menjelaskan skenario uji dari kebutuhan non fungsional.

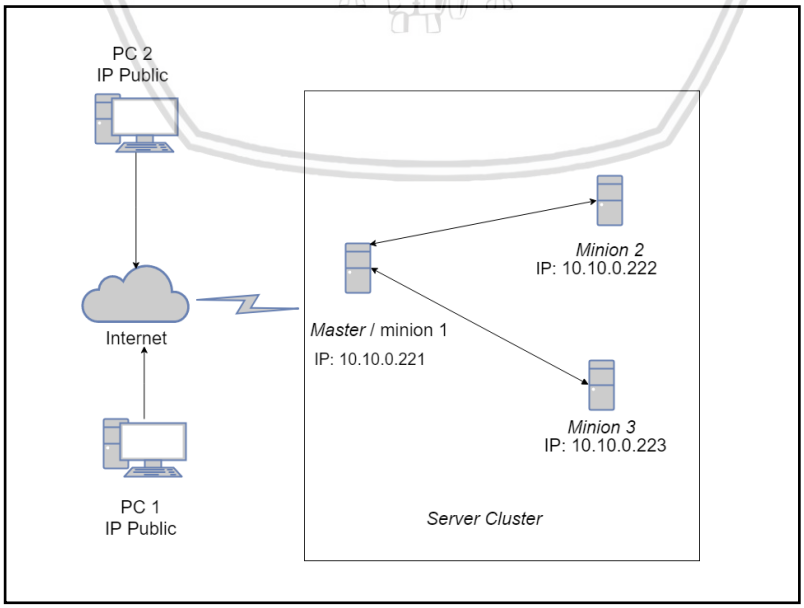
**Tabel 4.1 Rancangan skenario uji kebutuhan fungsional**

| | |
|---|---|
| Pengujian *web server* | Sistem akan diuji apakah mampu menjalankan layanan *web server.* |
| Pengujian *failure* | Pengujian ini akan dilakukan dengan dua cara yang berbeda. Pertama mematikan *node* yang menjalankan *web server* secara paksa*,* dan kedua mematikan layanan *web server* secara paksa. Tujuan dari pengujian ini untuk mengetahui apakah sistem mampu menyediakan layanan *web server* secara *high availability.* |
| Pengujian *autoscaling* | Pengujian ini bertujuan untuk mengetahui apakah sistem mampu melakukan *autoscaling* baik *scale up* dan *Scale down* pada parameter yang |

| | |
|---|---|
| | telah ditentukan secara otomatis dengan parameter *scaling* berupa CPU *usage.* |

**Tabel 4.2 Rancangan skenario uji kebutuhan non fungsional**

| | |
|---|---|
| Pengujian PING | Pengujian ini bertujuan apakah *server* didalam cluster dapat berkomunikasi satu sama lain. |
| Pengujian waktu *failover* | Pengujian ini bertujuan untuk mengetahui berapa rata-rata waktu yang dibutuhkan untuk melakukan *failover web server.* |
| Pengujian waktu *autoscaling* | Pengujian ini bertujuan untuk mengetahui berapa rata-rata waktu yang dibutuhkan oleh sistem untuk melakukan *scale up* dan *Scale down web server.* |
| Pengujian kinerja CPU *usage* | Pengujian ini bertujuan untuk mengetahui apakah *autoscaling* mempengaruhi kinerja *autoscaling,* dan berapa besar pengaruh *autoscaling* terhadap CPU *usage.* |




**Gambar 4.2 *Topology* pengujian**

Gambar 4.2 Menunjukan *Topology* yang digunakan pada pengujian sistem. Pada pengujian tersebut menunjukan bahwa sistem akan diakses dari luar *cluster* menggunakan pc yang telah disediakan IP dari PC adalah IP *public* dimana IP tersebut dapat diakses langsung dari internet. Pada pengujian *web server* PC 1 dan 2 akan mengakses IP dari *server* yang memiliki layanan *web server* untuk mengetahui apakah layanan telah tersedia. Pada pengujian *autoscaling* PC 1 dan 2 akan mengirim *request* secara berulang kepada *server* untuk menaikan konsumsi CPU. Pada pengujian PING *master node* didalam *cluster* akan mengirimkan PING kepada *minion* 1 dan 2 untuk mengetahui apakah *server* telah tersambung.

## 4.5 Implementasi

Implementasi sistem bertujuan untuk merealisasikan perancangan yang telah dibuat sebelumnya dimana sistem yang dibuat akan diuji apakah dapat menyelesaikan masalah yang ada dan bagaimana performa dari sistem tersebut. Sistem yang dibangun akan berjalan pada lingkup virtual berbentuk *clustering* dimana perangkat keras terdapat lebih dari satu namun terletak pada network yang sama karena hanya bersifat prototipe sehingga tidak mengganggu sistem lain yang berjalan.

### 4.5.1 Spesifikasi perangkat keras

Masing-masing *node* memiliki spesifikasi seperti pada tabel 4.3

**Tabel 4.3 kebutuhan perangkat keras**

| | |
|---|---|
| Prosesor | *single core* |
| RAM | 2 GB |
| *Storage* | 50 GB |

### 4.5.2 Spesifikasi perangkat lunak

- Sistem Operasi Linux
- Docker
- Kubernetes
- NGINX
- Linux KVM
- Putty untuk melakukan remote access
- Kubespray
- Ansible

### 4.5.3 Spesifikasi jaringan

Jaringan yang digunakan pada *cluster* adalah Calico sebagai *overlay network* dimana proses bertukar data seperti *scheduling, job,* dan komunikasi antar *master* dan *minion* terjadi. Tabel 4.4 akan menjelaskan pembagian IP pada masing-masing *node.*

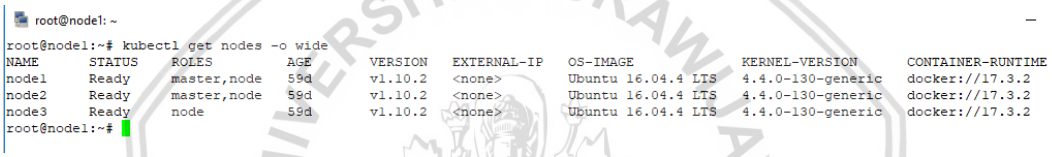
**Tabel 4.4 Pembagian IP *Node***

| | |
|---|---|
| *Master / Node 1* | 10.10.0.221/24 |
| *Node 2* | 10.10.0.222/24 |
| *Node 3* | 10.10.0.223/24 |

Pada tabel 4.4 menunjukan IP yang digunakan oleh masing-masing *node*, setiap *node* berada pada *network* yang sama yaitu 10.10.0.0. *Netmask* yang digunakan adalah /24 dimana hal ini secara default diberikan oleh sistem *server* sesuai ketersediaan yang ada.

### 4.5.4 Konfigurasi Kubernetes

Untuk melakukan konfigurasi sistem, *tools* Putty digunakan untuk melakukan *remote connection*. Kubernetes akan di-*deploy* dan dikonfigurasi menggunakan Kubespray. Semua *script* konfigurasi sistem menggunakan format yaml




**Gambar 4.3 Hasil dari *deployment* Kubernetes**

Gambar 4.4 menunjukan hasil dari deployment yang telah dijalankan, dapat dilihat bahwa *node1* sampai dengan *node3* telah terhubung kedalam *cluster*, beralan menggunakan Ubuntu 16.04.4 LTS dan Docker versi 17.3.2.

### 4.5.5 Konfigurasi layanan pada *cluster*

Pada bagian ini service dasar seperti Heapster untuk mengumpulkan data *metric* status dari *pod* dan *node,* InfluxDB untuk menyimpan data *metric,* dan *Web UI dashboard* untuk monitoring dan manajemen layanan akan di-*install* pada *node master.*

- **Konfigurasi *Monitoring***

  Pada layanan monitoring terdapat tiga komponen utama untuk monitoring cluster yaitu Heapster, InfluxDB, *Webdashboard.* Heapster berguna untuk mengumpulkan *metric* status dari layanan maupun node seperti kondisi CPU dan RAM, InfluxDB berguna untuk menyimpan data *metric* dari Heapster, sementara *webdashboard* merupakan UI *built in* yang disediakan oleh Kubernetes untuk proses monitoring dan manajemen *cluster.* Tabel 4.5 sampai 4.7 merupakan skrip konfigurasi yang digunakan untuk konfiguras dari layanan monitoring.

**Tabel 4.5 Konfigurasi heapster**

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: heapster
  namespace: kube-system
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: heapster
  namespace: kube-system
spec:
  replicas: 1
  template:
    metadata:
      labels:
        task: monitoring
        k8s-app: heapster
    spec:
      serviceAccountName: heapster
      containers:
      - name: heapster
        image: k8s.gcr.io/heapster-amd64:v1.5.3
        imagePullPolicy: IfNotPresent
        command:
        - /heapster
        - --source=kubernetes:https://kubernetes.default
        - --sink=influxdb:http://monitoring-influxdb.kube-
system.svc:8086
---
apiVersion: v1
kind: Service
metadata:
  labels:
    task: monitoring
    # For use as a Cluster add-on
(https://github.com/kubernetes/kubernetes/tree/master/clus
ter/addons)
    # If you are NOT using this as an addon, you should
comment out this line.
    kubernetes.io/cluster-service: 'true'
    kubernetes.io/name: Heapster
  name: heapster
  namespace: kube-system
spec:
  ports:
  - port: 80
    targetPort: 8082
  selector:
    k8s-app: heapster
```

**Tabel 4.6 Konfigurasi InfluxDB**

```
apiVersion: extensions/v1beta1
```

```
kind: Deployment
metadata:
  name: monitoring-influxdb
  namespace: kube-system
spec:
  replicas: 1
  template:
    metadata:
      labels:
        task: monitoring
        k8s-app: influxdb
    spec:
      containers:
      - name: influxdb
        image: k8s.gcr.io/heapster-influxdb-amd64:v1.3.3
        volumeMounts:
        - mountPath: /data
          name: influxdb-storage
      volumes:
      - name: influxdb-storage
        emptyDir: {}
---
apiVersion: v1
kind: Service
metadata:
  labels:
    task: monitoring
    # For use as a Cluster add-on
(https://github.com/kubernetes/kubernetes/tree/master/clus
ter/addons)
    # If you are NOT using this as an addon, you should
comment out this line.
    kubernetes.io/cluster-service: 'true'
    kubernetes.io/name: monitoring-influxdb
  name: monitoring-influxdb
  namespace: kube-system
spec:
  ports:
  - port: 8086
    targetPort: 8086
  selector:
    k8s-app: influxdb
```

**Tabel 4.7 Konfigurasi *webdashboard***

```
# Copyright 2017 The Kubernetes Authors.
#
# Licensed under the Apache License, Version 2.0 (the
"License");
# you may not use this file except in compliance with the
License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
```

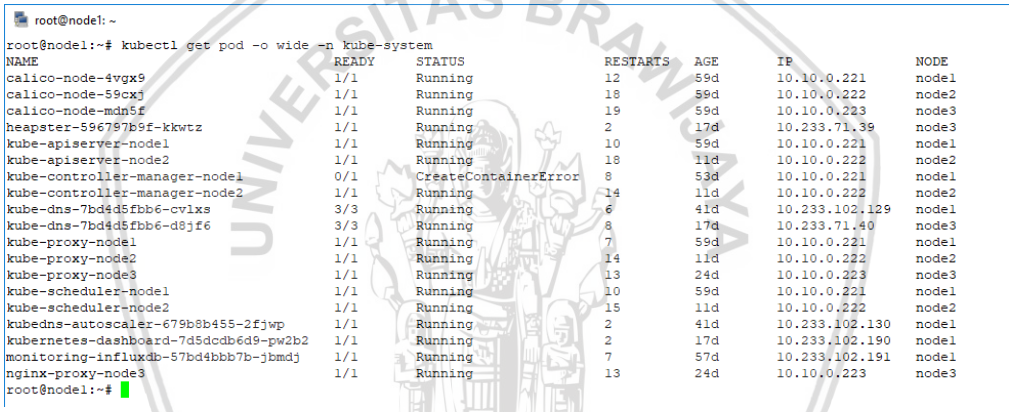
```
# Unless required by applicable law or agreed to in
writing, software
# distributed under the License is distributed on an "AS
IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
express or implied.
# See the License for the specific language governing
permissions and
# limitations under the License.

# Configuration to deploy release version of the Dashboard
UI compatible with
# Kubernetes 1.8.
#
# Example usage: kubectl create -f <this_file>

# ------------------- Dashboard Secret -------------------
#

apiVersion: v1
kind: Secret
metadata:
  labels:
    k8s-app: kubernetes-dashboard
  name: kubernetes-dashboard-certs
  namespace: kube-system
type: Opaque

---
# ------------------- Dashboard Service Account ----------
--------- #

apiVersion: v1
kind: ServiceAccount
metadata:
  labels:
    k8s-app: kubernetes-dashboard
  name: kubernetes-dashboard
  namespace: kube-system

---
# ------------------- Dashboard Role & Role Binding ------
------------ #

kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: kubernetes-dashboard-minimal
  namespace: kube-system
rules:
  # Allow Dashboard to create 'kubernetes-dashboard-key-
holder' secret.
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["create"]
  # Allow Dashboard to create 'kubernetes-dashboard-
settings' config map.
```

```
- apiGroups: [""]
  resources: ["configmaps"]
  verbs: ["create"]
  # Allow Dashboard to get, update and delete Dashboard
exclusive secrets.
- apiGroups: [""]
  resources: ["secrets"]
  resourceNames: ["kubernetes-dashboard-key-holder",
"kubernetes-dashboard-certs"]
  verbs: ["get", "update", "delete"]
  # Allow Dashboard to get and update 'kubernetes-
dashboard-settings' config map.
- apiGroups: [""]
  resources: ["configmaps"]
  resourceNames: ["kubernetes-dashboard-settings"]
  verbs: ["get", "update"]
  # Allow Dashboard to get metrics from heapster.
- apiGroups: [""]
  resources: ["services"]
  resourceNames: ["heapster"]
  verbs: ["proxy"]
- apiGroups: [""]
  resources: ["services/proxy"]
  resourceNames: ["heapster", "http:heapster:",
"https:heapster:"]
  verbs: ["get"]

---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: kubernetes-dashboard-minimal
  namespace: kube-system
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: kubernetes-dashboard-minimal
subjects:
- kind: ServiceAccount
  name: kubernetes-dashboard
  namespace: kube-system

---
# ------------------- Dashboard Deployment ---------------
---- #

kind: Deployment
apiVersion: apps/v1beta2
metadata:
  labels:
    k8s-app: kubernetes-dashboard
  name: kubernetes-dashboard
  namespace: kube-system
spec:
  replicas: 1
  revisionHistoryLimit: 10
  selector:
```

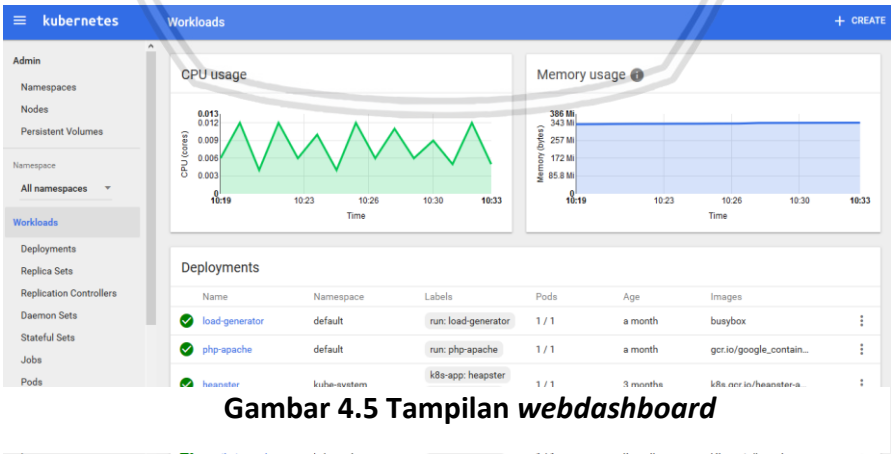
```
    matchLabels:
      k8s-app: kubernetes-dashboard
  template:
    metadata:
      labels:
        k8s-app: kubernetes-dashboard
    spec:
      containers:
      - name: kubernetes-dashboard
        image: k8s.gcr.io/kubernetes-dashboard-
amd64:v1.8.3
        ports:
        - containerPort: 8443
          protocol: TCP
        args:
          - --auto-generate-certificates
          # Uncomment the following line to manually
specify Kubernetes API server Host
          # If not specified, Dashboard will attempt to
auto discover the API server and connect
          # to it. Uncomment only if the default does not
work.
          # - --apiserver-host=http://my-address:port
        volumeMounts:
        - name: kubernetes-dashboard-certs
          mountPath: /certs
          # Create on-disk volume to store exec logs
        - mountPath: /tmp
          name: tmp-volume
        livenessProbe:
          httpGet:
            scheme: HTTPS
            path: /
            port: 8443
          initialDelaySeconds: 30
          timeoutSeconds: 30
      volumes:
      - name: kubernetes-dashboard-certs
        secret:
          secretName: kubernetes-dashboard-certs
      - name: tmp-volume
        emptyDir: {}
      serviceAccountName: kubernetes-dashboard
      # Comment the following tolerations if Dashboard
must not be deployed on master
      tolerations:
      - key: node-role.kubernetes.io/master
        effect: NoSchedule

---
# ------------------- Dashboard Service ------------------
- #

kind: Service
apiVersion: v1
metadata:
  labels:
```

```
    k8s-app: kubernetes-dashboard
  name: kubernetes-dashboard
  namespace: kube-system
spec:
  ports:
    - port: 443
      targetPort: 8443
  selector:
    k8s-app: kubernetes-dashboard
```

Untuk menjalankan skrip maka syntax yang digunakan adalah “kubectl create -f <nama skrip>”, lalu untuk melihat *pod* atau proses yang berhasil dibuat dapat dijalankan perintah “kubectl get pod -o wide –namespace=kube-system” dimana sistem akan menampilkan semua layanan yang berjalan pada *namespace* kube-system




**Gambar 4.4 Status layanan *monitoring***




**Gambar 4.5 Tampilan *webdashboard***

Gambar 4.5 menunjukan kondisi dari layanan monitoring yang berhasil berjalan serta node dimana layanan tersebut berjalan, lalu IP berarti cluster ip yang didapata layanan tersebut untuk berkomunikasi didalam cluster. Gambar 4.6 merupakan tampilan dari webdashboard dimana berfungsi untuk menampilkan

kondisi dari cluster memanajemne cluster seperti menghapus, menambah, dan mengubah pod, deployment, replica dari cluster.

### 4.5.6 Konfigurasi layanan *webserver* NGINX dan *failover*

Konfigurasi *webserver* NGINX menggunakan *images* yang diunduh dari Dockerhub sehingga *webserver* dapat digunakan sesuai kebutuhan dan dapat dihapus jika tidak diperlukan lagi. Tabel 4.8 menunjukan *script* konfigurasi dari *webserver.*

**Tabel 4.8 Konfigurasi *web server* dan *failover***

```
apiVersion: apps/v1 # for versions before 1.9.0 use
apps/v1beta2
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9 ports:
        - containerPort: 80
        resources:
         requests:
          memory: "64Mi"
          cpu: "250m"
         limits:
          memory: "128Mi"
          cpu: "1"
```

Pada tabel 4.8 merupakan konfigurasi dari layanan *webserver*, *images* yang digunakan merupakan NGINX versi 1.7.9 yang diunduh dari Dockerhub dimana port yang digunakan adalah port 80 pada kontainer, *replicas* akan memastikan

bahwa layanan *web server* akan dijalankan sebanyak satu *instance*, lalu pada bagian *resource* terdapat deklarasi *memory* dan *CPU* dimana layanan akan me-*request* sumber daya sebanyak 64Mi *memory,* dan 250m *CPU* lalu sistem akan membatasi *resource* sebanyak 128Mi *memory*, dan satu *core CPU*.

### 4.5.7 Konfigurasi *autoscaling*

Pada proses *auto scaling* Kubernetes memiliki fitur yang dapat melakukan *scaling* secara horizontal dimana Kubernetes akan memonitoring setiap layanan seperti *cpu* dan *ram*. Pada penelitian ini parameter layanan yang digunakan adalah *cpu* dimana kondisi *cpu* akan diset pada tingkat 20% untuk memudahkan proses pengujian. Berikut konfigurasi dari *autoscaling*

**Tabel 4.9 Konfigurasi *autoscaling***

```
$ kubectl autoscale deployment <nama deployment> --
cpu-percent=<target cpu> --min=<jumlah minimal pod>
--max=<jumlah maksimal pod>
```

Perintah pada tabel 4.9 dijalankan melalui terminal dengan menggunaka perintah kubectl. –min berarti jumlah minimal pod yang dijalankan jika *cpu* belum mencapai parameter yang ditentukan, --max adalah jumlah maksimal dari layanan yang akan dijalankan jika *cpu* melebihi jumlah yang ditentukan, <nama deployment> adalah nama layanan yang akan dimonitoring oleh Kubernetes, dan <target cpu> adalah berapa target dari utilisasi *cpu* yang diinginkan. Kubernetes akan mendtekesi dan menjalankan layanan secara otomatis agar parameter yang telah ditentukan tercapai.

# BAB 5 PENGUJIAN DAN ANALISIS HASIL

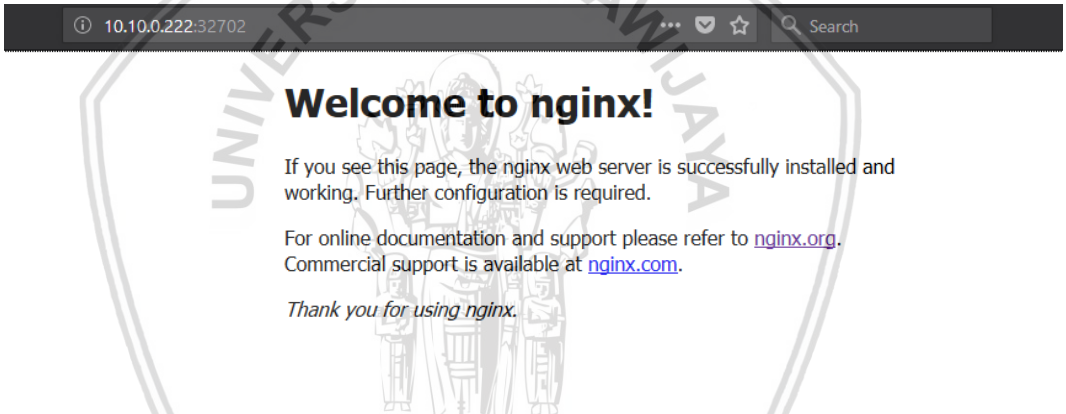
Pada bab ini dilakukan pengujian pada sistem yang bertujuan untuk mengetahui fungsionalitas dan kinerja dari sistem, serta menganalisis bagaimana hasil dari pengujian yang dilakukan. Pengujian akan dilakukan pada kebutuhan fungsional dan non fungsional.

## 5.1 Pengujian fungsional

### 5.1.1 Pengujian layanan *web server*

Pada pengujian ini layanan *web server* akan di-*deploy* menggunakan Kubernetes untuk menguji apakah layanan berhasil berjalan pada *cluster server*. *source code* yang digunakan sesuai pada tahap implementasi dan akan diakses menggunakan *browser* pada jaringan yang sama. *Web server* yang akan digunakan adalah NGINX.

### 5.1.2 Analisis hasil layanan *web server*



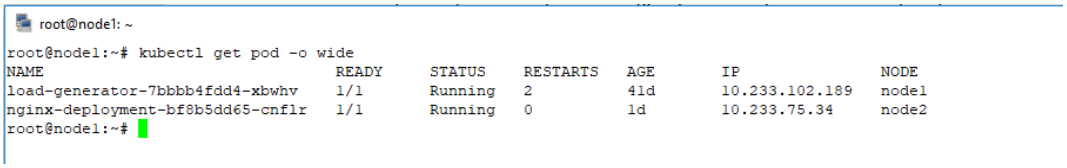

**Gambar 5.1 Layanan *web server* NGINX pada *browser***

Pada gambar 5.1 menunjukan bahwa layanan NGINX telah berhasil berjalan pada *node* 2 10.10.0.222 dengan *port* 32702 dimana port tersebut telah dialokasikan khusus untuk layanan NGINX.

### 5.1.3 Pengujian *failover*

Pada pengujian ini sistem akan diuji apakah mampu melakukan *failover web server* dengan dua cara pertama mematikan layanan *web server* secara paksa, dan kedua mematikan *node* yang memiliki web server secara paksa.

- **Mematikan layanan *web server* secara paksa**




**Gambar 5.2 Status awal layanan NGINX**

```
root@node1:~# kubectl get pod -o wide
NAME                                READY     STATUS              RESTARTS   AGE       IP              NODE
load-generator-7bbbb4fdd4-xbwhv     1/1       Running             2          41d       10.233.102.189  node1
nginx-deployment-bf8b5dd65-5cgn2    0/1       ContainerCreating   0          2s        <none>          node1
nginx-deployment-bf8b5dd65-cnflr    1/1       Terminating         0          1d        10.233.75.34    node2
root@node1:~#
```

**Gambar 5.3 Proses *failover* layanan NGINX**

Gambar 5.2 menunjukan status awal dari layanan NGINX dapat dilihat bahwa NGINX berjalan pada *node2.* Dengan perintah "kubectl delete pod<nama layanan>" untuk menghapus layanan maka sistem akan secara otomatis membuat layanan baru pada *node* lain. Status Terminating pada gambar 5.3 menunjukan bahwa *web server* pada *node2* telah dimatikan, lalu *web server* pindah pada *node1.*

- **Mematikan *node web server* secara paksa**

Pada proses ini *node* yang memiliki layanan *web server* NGINX akan dimatikan secara paksa.

```
root@node1:~# kubectl get pod -o wide
NAME                                READY     STATUS    RESTARTS   AGE       IP              NODE
load-generator-7bbbb4fdd4-xbwhv     1/1       Running   2          41d       10.233.102.189  node1
nginx-deployment-bf8b5dd65-h4nvh    1/1       Running   0          1m        10.233.75.33    node2
root@node1:~#
```

**Gambar 5.5 Status awal NGINX**

```
NAME                                READY     STATUS    RESTARTS   AGE       IP              NODE
load-generator-7bbbb4fdd4-xbwhv     1/1       Running   2          41d       10.233.102.189  node1
nginx-deployment-bf8b5dd65-gkd7h    1/1       Running   0          34s       10.233.71.41    node3
nginx-deployment-bf8b5dd65-h4nvh    1/1       Unknown   0          6m        10.233.75.33    node2
root@node1:~#
```

**Gambar 5.4 Proses *failover* NGINX**

Gambar 5.4 menunjukan status awal dari layanan NGINX yang berjalan pada *node2,* kemudian *node* dimatikan secara paksa sehingga proses *failover* pada gambar 5.5 berjalan pada gambar tersebut *web server* telah berhasil berpindah pada *node 3* dan *web server* pada *node2* mengalami status *unknown* hal ini terjadi karena *node2* tidak sempat mengirim status pada *node master* sehingga *node*

```
NAME                                READY     STATUS    RESTARTS   AGE       IP              NODE
load-generator-7bbbb4fdd4-xbwhv     1/1       Running   2          41d       10.233.102.189  node1
nginx-deployment-bf8b5dd65-gkd7h    1/1       Running   0          4m        10.233.71.41    node3
root@node1:~#
```

**Gambar 5.6 Status NGINX pada saat node2 dihidupkan kembali**

*master* tidak mendapatkan informasi apa-apa dari *node* tersebut. Ketika *node2* kembali dihidupkan makan *node* tersebut akan mengirim informasi kepada *node master* sehingga status *unknown* akan hilang dan layanan NGINX pada *node2* akan hilang dari daftar status seperti pada gambar 5.6.

### 5.1.4 Analisis hasil *failover*

Pada percobaan ini proses *failover* pada *web server* berhasil dilakukan baik ketika sistem mengalami *node failure* atau *service failure*. Terdapat perbedaan

ketika terjadi *node failure* sistem akan memberikan status *unknown* karena ketika *node* mati maka modul *kubelet* pada *node web server* tidak dapat mengirim info kepada *node master.* Proses *failover* pada saat *service failure* pun lebih cepat dibanding *node failure.*

### 5.1.5 Pengujian *autoscaling*

Pada pengujian ini sistem akan diuji apakah mampu menjalankan mekanisme *autoscaling* baik *scale up* ataupun *Scale down* secara otomatis, sistem akan diberi *request* pada *web server* hingga melebihi parameter CPU *usage*.

```
NAME                                READY     STATUS    RESTARTS   AGE       IP              NODE
load-generator-7bbbb4fdd4-xbwhv     1/1       Running   2          41d       10.233.102.189  node1
nginx-deployment-bf8b5dd65-gkd7h    1/1       Running   0          4m        10.233.71.41    node3
root@node1:~#
```

**Gambar 5.8 Status awal layanan NGINX**

```
root@node1:~# kubectl get hpa
NAME               REFERENCE                     TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
nginx-deployment   Deployment/nginx-deployment   0%/10%    1         3         1          59s
root@node1:~#
```

**Gambar 5.7 Status awal CPU *usage***

Gambar 5.7 dan Gambar 5.8 menunjukan kondisi awal dari sistem dimana CPU *usage* masih 0% karena belum terdapat *request* serta NGINX berjalan pada *node3.*

```
root@node1:~# kubectl get pod -o wide -w
NAME                                READY     STATUS    RESTARTS   AGE       IP              NODE
load-generator-7bbbb4fdd4-xbwhv     1/1       Running   2          41d       10.233.102.189  node1
nginx-deployment-bf8b5dd65-gkd7h    1/1       Running   0          18m       10.233.71.41    node3
nginx-deployment-bf8b5dd65-gqm5x    1/1       Running   0          32s       10.233.75.35    node2
nginx-deployment-bf8b5dd65-qfcn7    1/1       Running   0          32s       10.233.102.133  node1
```

**Gambar 5.9 Proses *autoscaling***

```
root@node1:~# kubectl get hpa -w
NAME               REFERENCE                     TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
nginx-deployment   Deployment/nginx-deployment   0%/10%    1         3         1          6m
nginx-deployment   Deployment/nginx-deployment   26%/10%   1         3         1          7m
nginx-deployment   Deployment/nginx-deployment   26%/10%   1         3         3          8m
nginx-deployment   Deployment/nginx-deployment   20%/10%   1         3         3          8m
nginx-deployment   Deployment/nginx-deployment   20%/10%   1         3         3          9m
nginx-deployment   Deployment/nginx-deployment   10%/10%   1         3         3          9m
nginx-deployment   Deployment/nginx-deployment   10%/10%   1         3         3          10m
nginx-deployment   Deployment/nginx-deployment   10%/10%   1         3         3          10m
```

**Gambar 5.10 CPU *usage web server***

### 5.1.6 Analisis hasil *autoscaling*

Gambar 5.9 menunjukan proses *scale up web server* telah berhasil, dimana sistem menambahkan dua *web server* tambahan untuk membantu menangani *request* yang ada, pada gambar 5.10 terlihat bahwa pada saat CPU *usage*

mencapai 26% sistem akan langsung menjalakan mekanisme *autoscaling* dan membagi jalur traffic secara *round robin,* sehingga CPU *usage* berkurang menjadi 20% bahkan 10% hal ini menunjukan bahwa *autoscaling* dapat mempengaruhi kinerja CPU. Pada percobaan ini parameter CPU *usage* yang digunakan adalah 10%.

## 5.2 Pengujian non fungsional

### 5.2.1 Pengujian koneksi antar *node*

Pada pengujian ini akan dilakukan uji komunikasi antar *node* untuk mengetahui apakah node telah *up* dan terhubung kedalam *cluster*. Pengujian dilakukan dengan cara *node master* melakukan ping pada setiap *node slave.* Untuk melihat *node* telah *up* dan berjalan dapat menggunakan perintah "kubectl get nodes" pada *node* master.

```
NAME      STATUS    ROLES          AGE       VERSION
node1     Ready     master,node    59d       v1.10.2
node2     Ready     master,node    59d       v1.10.2
node3     Ready     node           59d       v1.10.2
root@node1:~#
```

**Gambar 5.11 Status awal *node***

```
PING 10.10.0.222 (10.10.0.222) 56(84) bytes of data.
64 bytes from 10.10.0.222: icmp_seq=1 ttl=64 time=0.353 ms
64 bytes from 10.10.0.222: icmp_seq=2 ttl=64 time=0.250 ms
64 bytes from 10.10.0.222: icmp_seq=3 ttl=64 time=0.264 ms
64 bytes from 10.10.0.222: icmp_seq=4 ttl=64 time=0.330 ms
64 bytes from 10.10.0.222: icmp_seq=5 ttl=64 time=0.351 ms
^C
--- 10.10.0.222 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4003ms
rtt min/avg/max/mdev = 0.250/0.309/0.353/0.047 ms
root@node1:~#
```

**Gambar 5.13 Hasil ping *node2***

```
root@node1:~# ping 10.10.0.223
PING 10.10.0.223 (10.10.0.223) 56(84) bytes of data.
64 bytes from 10.10.0.223: icmp_seq=1 ttl=64 time=0.535 ms
64 bytes from 10.10.0.223: icmp_seq=2 ttl=64 time=0.315 ms
64 bytes from 10.10.0.223: icmp_seq=3 ttl=64 time=0.333 ms
64 bytes from 10.10.0.223: icmp_seq=4 ttl=64 time=0.405 ms
64 bytes from 10.10.0.223: icmp_seq=5 ttl=64 time=0.275 ms
^C
--- 10.10.0.223 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4003ms
rtt min/avg/max/mdev = 0.275/0.372/0.535/0.093 ms
root@node1:~#
```

**Gambar 5.12 Hasil ping *node3***

### 5.2.2 Analisis hasil koneksi antar *node*

Pada gambar 5.11 menunjukan bahwa terdapat 3 *node* yang telah tersedia. Gambar 5.12 dan 5.13 menunjukan bahwa setiap node telah *up* dan berhasil berkomunikasi dengan *node master* didalam *cluster*.

### 5.2.3 Pengujian waktu *failover*

Pada pengujian ini sistem akan diuji dengan dua skenario uji. Pada skenario pertama *node* yang memiliki layanan *webserver* akan dimatikan secara paksa sehingga sistem mengalami *failure.* Pada skenario uji kedua layanan *web server* akan dimatikan secara paksa. pengujian dilakukan sebanyak lima kali untuk mendapatkan rata-rata waktu yang dibutuhkan untuk melakukan *failover.* Dengan keterbatasan sumber daya maka hanya satu layanan *web server* yang akan digunakan. Untuk melihat detail waktu digunakan *log* dari kubernetes menggunakan perintah "*kubectl describe* <nama pod/proses>" dan "*kubectl describe* <nama *node*>". Pada gambar 5.14 menunjukan *status log deployment* seperti waktu mulai, *images* yang digunakan, *node* tempat deployment dijalankan dan keterangan lainnya, sementara gambar 5.15 menunjukan *log* dari *node*.

```
root@node1:~# kubectl describe pod nginx-deployment-bf8b5dd65-gkd7h
Name:           nginx-deployment-bf8b5dd65-gkd7h
Namespace:      default
Node:           node3/10.10.0.223
Start Time:     Sat, 04 Aug 2018 17:07:27 +0700
Labels:         app=nginx
                pod-template-hash=694618821
Annotations:    <none>
Status:         Running
IP:             10.233.71.41
Controlled By:  ReplicaSet/nginx-deployment-bf8b5dd65
Containers:
  nginx:
    Container ID:   docker://07df6e324493c87a2fe81da80fa47922fb3de6574e792819cc4d83ef6f2cbd05
    Image:          nginx:1.7.9
    Image ID:       docker-pullable://nginx@sha256:e3456c851a152494c3e4ff5fcc26f240206abac0c9d794affb40e0714846c451
    Port:           80/TCP
    Host Port:      0/TCP
    State:          Running
      Started:      Sat, 04 Aug 2018 17:07:31 +0700
    Ready:          True
    Restart Count:  0
    Limits:
      cpu:     1
      memory:  128Mi
    Requests:
      cpu:        250m
      memory:     64Mi
    Environment:  <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-hztkc (ro)
Conditions:
  Type           Status
  Initialized    True
  Ready          True
  PodScheduled   True
Volumes:
  default-token-hztkc:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  default-token-hztkc
    Optional:    false
QoS Class:       Burstable
Node-Selectors:  <none>
Tolerations:     <none>
```

**Gambar 5.14 Status dari Kubectl *get event***

```
root@node1:~# kubectl describe nodes node3
Name:               node3
Roles:              node
Labels:             beta.kubernetes.io/arch=amd64
                    beta.kubernetes.io/os=linux
                    kubernetes.io/hostname=node3
                    node-role.kubernetes.io/node=true
Annotations:        node.alpha.kubernetes.io/ttl=0
                    volumes.kubernetes.io/controller-managed-attach-detach=true
CreationTimestamp:  Tue, 05 Jun 2018 21:46:08 +0700
Taints:             <none>
Unschedulable:      false
Conditions:
  Type             Status  LastHeartbeatTime                 LastTransitionTime                Reason                       Message
  ----             ------  -----------------                 ------------------                ------                       -------
  OutOfDisk        False   Sat, 04 Aug 2018 17:38:09 +0700   Tue, 10 Jul 2018 18:54:02 +0700   KubeletHasSufficientDisk     kubelet has sufficient disk space available
  MemoryPressure   False   Sat, 04 Aug 2018 17:38:09 +0700   Tue, 10 Jul 2018 18:54:02 +0700   KubeletHasSufficientMemory   kubelet has sufficient memory available
  DiskPressure     False   Sat, 04 Aug 2018 17:38:09 +0700   Tue, 10 Jul 2018 18:54:02 +0700   KubeletHasNoDiskPressure     kubelet has no disk pressure
  PIDPressure      False   Sat, 04 Aug 2018 17:38:09 +0700   Tue, 05 Jun 2018 21:46:08 +0700   KubeletHasSufficientPID      kubelet has sufficient PID available
  Ready            True    Sat, 04 Aug 2018 17:38:09 +0700   Tue, 10 Jul 2018 18:54:02 +0700   KubeletReady                 kubelet is posting ready status. AppArmor en
abled
Addresses:
```

**Gambar 5.15 Status dari Kubectl *describe***

### 5.2.4 Hasil pengujian waktu *failover*

Berikut data yang didapatkan dari pengujian sistem. Tabel 5.1 menunjukan hasil rata-rata waktu pengujian ketika *node* dimatikan secara paksa, Tabel 5.2 menunjukan hasil rata-rata waktu ketika layanan *web server* dimatikan secara paksa.

**Tabel 5.1 Hasil pengujian *failover node failure***

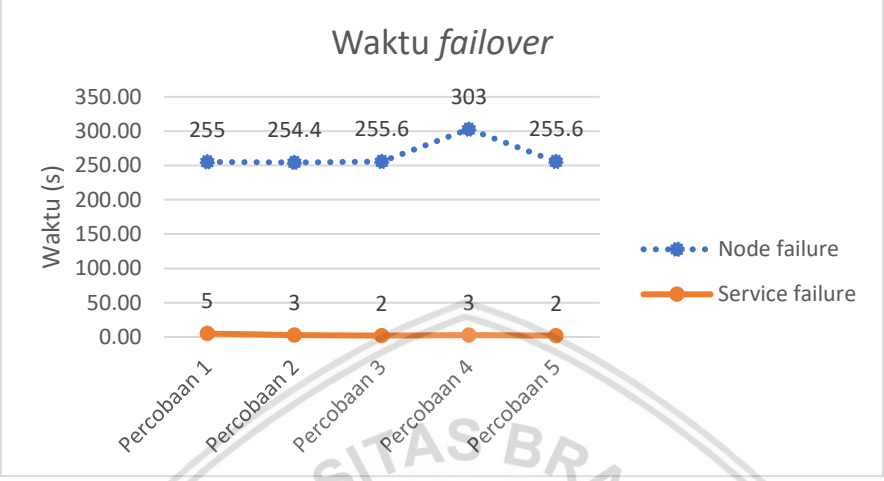
| Pengujian | Waktu *failover (s)* |
|---|---|
| 1 | 255 |
| 2 | 254.4 |
| 3 | 255.6 |
| 4 | 303 |
| 5 | 255.6 |
| **Rata-rata** | 264.74 |

**Tabel 5.2 Hasil pengujian *failover service failure***

| Pengujian | Waktu *failover (s)* |
|---|---|
| 1 | 5 |
| 2 | 3 |
| 3 | 2 |
| 4 | 3 |
| 5 | 2 |

| | |
|---|---|
| **Rata-rata** | 3 |

### 5.2.5 Analisis hasil waktu *failover*




**Gambar 5.16 Grafik waktu *failover***

Gambar 5.16 menunjukan waktu yang dibutuhkan untuk melakukan *failover* pada sebuah layanan *web server*. Rata-rata waktu yang dibutuhkan adalah 264.74s, atau 4 menit 41 detik untuk *node failure* dan rata-rata waktu yang dibutuhkan untuk *service failure* adalah 3 *second.* Proses *failover* pada pengujian *service failure* memiliki rata-rata waktu yang lebih singkat hal ini disebabkan ketika *node* dimatikan secara paksa modul *kubelet* pada node tidak dapat berkomunikasi dengan *node master* sehingga *node master* membutuhkan waktu lebih lama untuk melakukan pengecekan, semantara pada pengujian *service failure* ketika hanya layanan *web server* saja yang dimatikan maka *kubelet* akan mengirimkan informasi kepada *node master* untuk segera menyediakan layanan *web server* baru.

### 5.2.6 Pengujian *autoscaling*

Pada pengujian ini akan diuji berapa rata-rata waktu yang dibutuhkan oleh sistem untuk melakukan *scale up* dan *Scale down* sesuai parameter batas CPU yang ditentukan secara otomatis, serta berapa CPU *usage* yang digunakan.

### *5.2.7 Scale up*

Pengujian ini bertujuan untuk mengetahui berapa rata-rata waktu yang dibutuhkan sistem untuk melakukan *scale up* layanan pada sistem. Parameter CPU yang digunakan adalah sebesar 10% dari CPU yang dialokasikan pada layanan *web server* serta maksimala layanan adalah 3 dan minimal 1 *web server*.

### 5.2.8 Hasil pengujian *scale up*

**Tabel 5.3 Hasil pengujian *scale up***

| Percobaan | Waktu (s) |
|---|---|
| 1 | 60 |
| 2 | 60 |
| 3 | 30 |
| 4 | 45 |
| 5 | 30 |
| **Rata-rata** | 45 |

Tabel 5.3 menunjukan rata-rata waktu yang dibutuhkan oleh sistem dalam melakukan *scale up* layanan *web server* sistem akan mendeteksi apakah CPU *usage* telah mencapai parameter yang ditentukan, jika iya maka sistem akan melakukan *scaling* secara otomatis sesuai kebutuhan sistem dan *resource* yang dimiliki.

### 5.2.9 *Scale down*

Pengujian ini bertujuan untuk mendapatkan rata-rata waktu yang dibutuhkan untuk *Scale down* sebuah layanan. Sistem yang diuji memiliki parameter CPU *usage* sebesar 10% dari CPU yang dimiliki oleh layanan, jumlah maksimal layanan yang digunakan adalah 3 dan minimal 1 *web server.*

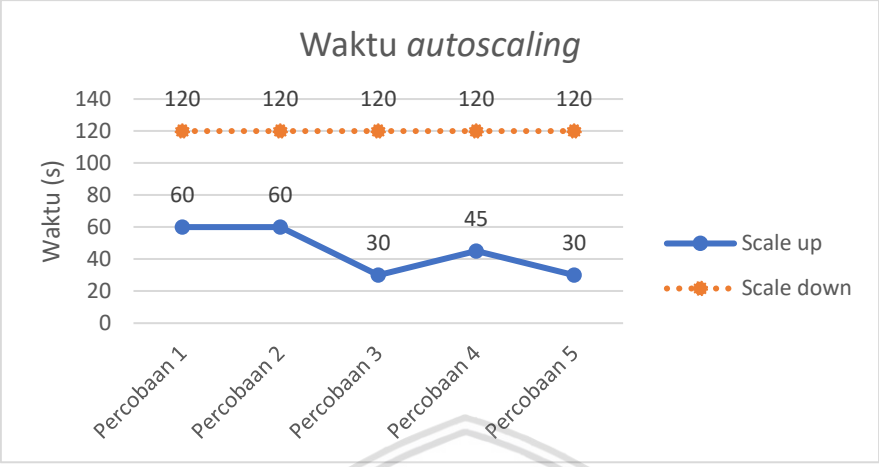
### 5.2.10 Hasil pengujian *Scale down*

**Tabel 5.4 Hasil pengujian *Scale down***

| Percobaan | Waktu (s) |
|---|---|
| 1 | 120 |
| 2 | 120 |
| 3 | 120 |
| 4 | 120 |
| 5 | 120 |
| **Rata-rata** | 120 |

Table 5.4 menunjukan hasil dari waktu rata-rata yang dibutuhkan sistem untuk melakukan *Scale down* pada layanan *web server*.

### 5.2.11 Analisis hasil *scaling*




**Gambar 5.17 Grafik waktu *autoscaling***

Gambar 5.17 menunjukan hasil perbandingan waktu yang dibutuhkan untuk melakukan *scale up* maupun *Scale down* secara otomatis. Pada proses *Scale down* sistem membutuhkan waktu lebih lama dibanding *scale up*, karena sistem akan memberikan jeda waktu untuk melakukan *Scale down* hal ini menjaga terjadinya lonjakan *traffic* baru sehingga sistem tidak perlu melakukan *scale up* ulang. Rata-rata waktu yang dibutuhkan pada proses *Scale down adalah* 120s, dan *scale up* sebesar 45s. Setelah proses *scaling* dilakukan maka Kubernetes akan membagi *traffic* kepada layanan *web server* baru pada proses *scale up,* atau membersihkan layanan *web server* pada proses *Scale down.*

### 5.2.12 CPU usage

Pada percobaan ini akan dilihat apakah *autoscaling* dapat mempengaruhi CPU *usage* dari *server* layanan. Monitoring CPU dilakukan pada *server* yang menjalankan *web server* pertama kali, lalu *server* tersebut akan diberikan *request* secara berulang hingga CPU *usage* naik dan mencapai parameter *autoscaling*. Setelah itu dicek apakah *autoscaling* mempengaruhi CPU *usage* atau tidak, dan dilakukan perbandingan dengan layanan yang tidak menggunakan *autoscaling.*

### 5.2.13 Hasil pengujian CPU *usage*

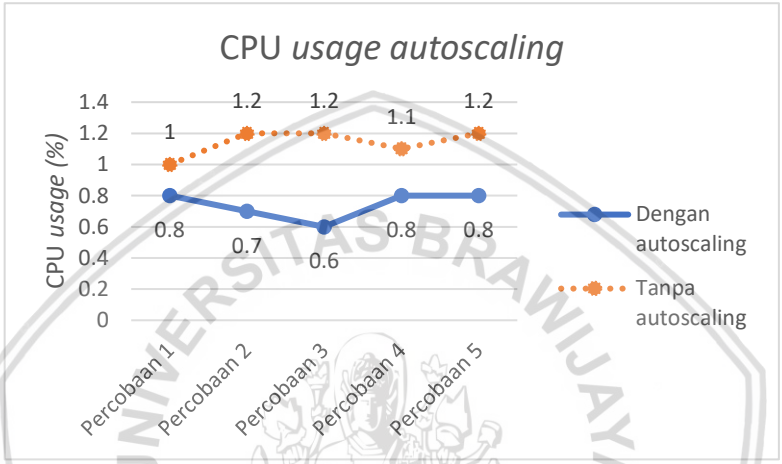
**Tabel 5.5 Hasil pengujian CPU *usage***

| Percobaan | CPU *usage* (%) | |
|---|---|---|
| | Dengan *autoscaling* | Tanpa *autoscaling* |
| 1 | 0.8 | 1 |
| 2 | 0.7 | 1.2 |
| 3 | 0.6 | 1.2 |

| | | |
|---|---|---|
| 4 | 0.8 | 1.1 |
| 5 | 0.8 | 1.2 |
| **Rata-rata** | 0.74 | 1.14 |

Tabel 5.5 menunjukan hasil CPU *usage* yang dibutuhkan oleh sistem selama pengujian *autoscaling web server.*

### 5.2.14 Analisis hasil CPU *usage*




**Gambar 5.18 Grafik CPU *usage***

Gambar 5.18 menunjukan CPU *usage* pada layanan yang menggunakan fitur *autoscaling* memiliki konsumsi *resource* yang lebih rendah 0.4% dibanding layanan yang tidak menggunakan, hal ini disebabkan saat CPU *usage* telah mencapai parameter yang ditentukan maka sistem akan melakukan *scale up* layanan *web server* lalu sistem akan membagi *traffic request* pada masing-masing layanan secara bergantian sehingga beban kerja pada satu *server* akan berkurang.

# BAB 6 PENUTUP

## 6.1 Kesimpulan

Dengan selesainya pelaksanaan penelitian ini didapat beberapa hasil dan analisis yang mampu menjawab latar belakang masalah yang telah dijabarkan. Hasil penelitin pun dapat menjawab rumusan masalah yang diberikan. Berikut adalah poin-poin kesimpulan yang didapat:

1. Perancangan dan implementasi layanan *web server* berbasis kontainer menggunakan Kubernetes dapat dilakukan. Implementasi sistem dilakukan pada lingkungan berbasis virtual sehingga mampu menghemat *resource* yang ada. Proses implementasi dilakukan dari persiapan perangkat keras, perangkat lunak, dan kontainer web server.
2. Perancangan dan implementasi *failover* dan *autoscaling* kontainer *web* server dapat dilakukan dengan Kubernetes. Proses perancangan dan implementasi dilakukan setelah Kubernetes, Docker, dan Kontainer *web server* berhasil di-*install,* lalu menyiapkan parameter *failover* dan *autoscaling.*
3. Hasil dari pengujian sistem menunjukan bahwa mekanisme *failover* mampu menjaga dan mem-*backup* layanan *web server* dari masalah *node* dan *service failure* dengan rata-rata waktu 264.74s untuk *node failure*, dan 3s untuk *service failure*. Mekanisme *autoscaling* pada Kubernetes membuat layanan *web server* dapat mengatasi lonjakan *traffic* yang tidak dapat ditangani oleh satu *web server* sehingga masing-masing *server* pada cluster mampu bekerja sama untuk melayani *request.* Hasil rata-rata waktu yang dibutuhkan untuk melakukan *scale up* adalah 45s, dan *Scale down* sebesar 120s. Penggunaan *autoscaling* mampu membuat CPU *usage* dari *server* berkurang sebesar 0.4% sehingga sumber daya *server* yang ada dapat lebih dimaksimalkan.

## 6.2 Saran

Berdasarkan penelitian yang telah dilaksanakan terdapat beberapa saran yang dapat dilakuakan pada penelitian berikutnya untuk lebih menyempurnakan penelitian selanjutnya:

1. Aplikasi yang dijalankan pada penelitian ini masih bersifat *stateless* dimana data yang terdapat pada sebuah layanan akan hilang ketika layanan tersebut hilang, dibutuhkan sebuah *persistent storage* yang mampu bersinkronisasi dengan semua server pada *cluster* untuk menyimpan data yang bersifat *statefull.*
2. Penelitian ini masih bersifat percobaan dimana semua komponennya diimplementasi pada lingkungan kerja virtual. Pada penelitian berikutnya diharapkan dapat diimplementasi pada perangakat keras *server* sebenarnya.

# DAFTAR PUSTAKA


Ansible. (2018). *Ansible*. Dipetik Juni 19, 2018, dari https://github.com/ansible

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., & Katz, R. (2009). *Above the Clouds: A Berkeley View of Cloud.* California: UC Berkeley Reliable Adaptive Distributed Systems Laboratory.

Bhowmik, S. (2017). *Cloud Computing.* India: Cambridge University Press.

Bigelow, S. J. (2015). *Five cons of container technology*. Dipetik Februari 16, 2018, dari http://searchservervirtualization.techtarget.com/feature/Five-cons-of-container-technology

Bondi, A. B. (2000). Characteristics of scalability and their impact on performance. *2nd international workshop on Software and performance* , 195-203.

Brooks, J. (2017, Agustus 22). *Atomic ContainerizedMaster*. Diambil kembali dari wiki.centos.org: https://wiki.centos.org/SpecialInterestGroup/Atomic/ContainerizedMaster

Docker. (2017, Agustus 15). *What is a Container*. Diambil kembali dari www.docker.io: https://www.docker.com/what-container

Durairaj, M., & Kannan, P. (2014). A Study On Virtualization Techniques And . *INTERNATIONAL JOURNAL OF SCIENTIFIC & TECHNOLOGY RESEARCH, 3*, 147.

Furht, B., & Escalante, A. (2010). *Handbook of Cloud Computing.* London: Springer Science & Business Media.

Gourley, D., & Totty, B. (2002). *HTTP: The Definitive Guide* (1st ed.). Sebastopol: O'reilly Media.

Grant, B. (2017, agustus 20). *WSO2Con US 2015 Kubernetes: a platform for automating deployment, scaling, and operations* . Diambil kembali dari www.slideshare.net: https://www.slideshare.net/BrianGrant11/wso2con-us-2015-kubernetes-a-platform-for-automating-deployment-scaling-and-operations

Gunawan, D., Basuki, A., & Bhawiyuga, A. (2016). APLIKASI ANTARMUKA PRAKTIKUM LABORATORIUM VIRTUAL JARINGAN KOMPUTER

BERBASISKAN OPENSTACK, NOVNC, DAN DOCKER. *Repositori Jurnal Mahasiswa PTIIK UB, 8*, 1-51.

Isabekyan, N. (2016). *Introduction to Kubernetes Architecture*. Dipetik Februari 16, 2018, dari https://x-team.com/blog/introduction-kubernetes-architecture/

Jayaswal, K. (2005). *Administering Data Centers: Servers, Storage, And Voice Over Ip*. new delhi: Wiley India Pvt. Limited.

Kopparapu, C. (2002). *Load Balancing Servers, Firewalls, and Caches*. canada: Wiley Computer Publishing.

Kubernetes. (2017, Agustus 21). *Kubernetes Components*. Diambil kembali dari www.kubernetes.io: https://kubernetes.io/docs/concepts/overview/components/

Kubernetes. (2017, agustus 8). *What is Kubernetes*. Diambil kembali dari kubernetes.io: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

Kubespray. (2018). *About*. Dipetik juni 18, 2018, dari http://kubespray.io/about/

Kusnetzky, D. (2011). *Virtualization: A Manager's Guide* (1st ed.). sebastopol: O'reilly Media Inc.

linux-kvm. (2018). *KVM*. Dipetik April 17, 2018, dari https://www.linux-kvm.org/page/Main_Page

Medel, V., Rana, O., Bañares, J. Á., & Arronategui, U. (2016). Modelling Performance & Resource Management in. *2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing*, 257.

Mouat, A. (2015). *Using Docker: Developing and Deploying Software with Containers*. sebastopol: O'Reilly Media, Inc.

NGINX. (2018). *Welcome to NGINX Wiki!* Dipetik maret 11, 2018, dari https://www.nginx.com/resources/wiki/

oracle. (2018). *Avoiding and Recovering From Server Failure*. Dipetik maret 3, 2018, dari https://docs.oracle.com/cd/E13222_01/wls/docs90/server_start/failures.html

Oracle. (2018). *Understanding WebLogic Server Clustering* . Dipetik maret 11, 2018, dari https://docs.oracle.com/cd/E11035_01/wls100/cluster/overview.html

Portnoy, M. (2016). *Virtualization Essentials* (2nd ed.). Canada: John Wiley & Sons.

*Project Atomic Getting Started Guide*. (2017, Agustus 21). Diambil kembali dari www.projectatomic.io: http://www.projectatomic.io/docs/gettingstarted/

Solichin, A. (2016). *Pemrograman Web dengan PHP dan MySQL* (1st ed.). Jakarta: Universitas Budi Luhur.

Suwastika, N. A., W, P. W., & Harsono, T. B. (2015). MODEL PREDIKSI SIMPLE MOVING AVERAGE PADA. *Jurnal Ilmiah Teknologi Informasi Terapan, I*, 37.

Syafrizal, M. (2005). *Pengantar Jaringan Komputer*. Yogyakarta: C.V. Andi Offset.

Tarkoma, S. (2010). *Overlay Networks: Toward Information Networking*. London: CRC Press.

Tarkoma, S. (2010). *Overlay Networks: Toward Information Networking*. London: CRC Press.

Turnbull, J. (2015). *The Docker Book* (1.6.0 ed.). lulu.com.

upcloud. (2017, Agustus 23). *How to Install HAProxy Load Balancer on CentOS*. Diambil kembali dari www.upcloud.com: https://www.upcloud.com/support/haproxy-load-balancer-centos/

Wardhana, P. W., & Suwastika, N. A. (2014). PERANCANGAN SISTEM AUTO-SCALING PADA CLOUD COMPUTING. *Seminar Nasional Teknologi Informasi dan Multimedia 2014*, 1-6.

webdevelopersnotes. (2018). *What is web server – a computer OR a program?* Dipetik Maret 12, 2018, dari https://www.webdevelopersnotes.com/what-is-web-server

www.haproxy.org. (2018). *haproxy*. Dipetik februari 16, 2018, dari http://www.haproxy.org/

Yuhefizar, Mooduto, H., & Hidayat, R. (2008). *Cara Mudah Membangun Website Interaktif Menggunakan Content Management System Joomla* (2nd ed.). Jakarta: Elex Media Komputindo.